PARALLEL TOKEN PROCESSING IN AN ASYNCHRONOUS TRIGGER SYSTEM

By

JONGBUM PARK

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1999

To my wife, *Sukhee Sung*

ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Dr. Eric Hanson, who helped me with careful guidance and personal understanding during last three and a half years. Also, many thanks to my Ph.D. committee members, Dr. Stanley Su, Dr. Sharma Chakravarthy, Dr. Richard Newman, and Dr. Howard Beck, who sacrificed their time and offered precious advice that helped me throughout my research.

I do not know how to express my thanks to my wife, Sukhee Sung, who had to spend many difficult and boring days in Gainesville. I cannot forget her ceaseless love and support until I die. I also thank to my daughter, Suwon, and my son, Charles, for not complaining too much about my neglect. I offer my families back in Korea my heartfelt thanks for their encouragement. In this writing, I cannot exclude Dr. John Penrod who voluntarily helped me with my English, and I respect him.

I really thank my sponsor, the Republic of Korea Air Force, who provided substantial financial and logistical support during my study in the United States. Finally, I would like to thank all the friends and colleagues who stood by my side and helped me whenever I needed it. I wish all of them good luck and divine protection.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

PARALLEL TOKEN PROCESSING IN AN ASYNCHRONOUS TRIGGER SYSTEM

By

Jongbum Park

May, 1999

Chairman: Eric N. Hanson
Major Department: Computer and Information Science and Engineering

In an asynchronous trigger system, trigger condition checking and trigger action execution are done after update transactions to data sources are completed. This research is motivated by the development of an asynchronous trigger processing and view maintenance system called *TriggerMan*. In *TriggerMan*, a data structure called a discrimination network is built to efficiently check the condition of a trigger. A *Gator* network is a kind of discrimination network.

In the first part of the research, we try to improve the performance of *TriggerMan* using techniques that do not compromise the semantic correctness of trigger processing. The techniques include *Gator* network dynamic restructuring, efficient processing of a large token set, parallel resource utilization, parallel processing of data, and parallel processing of trigger conditions and actions. However, parallel token processing causes problems in the semantic correctness of trigger processing.

In the second part of the research, to incorporate parallel token processing into an asynchronous trigger system in a productive way, we introduce four *consistency level*s of trigger processing. The purpose of *consistency level* is to achieve maximum performance for a given degree of semantic consistency. The lower the level is, the higher the performance is. Four consistency levels can be summarized as follows: Level 3 provides serial token processing semantics for a trigger, Level 2 executes trigger actions using all and only correct data, Level 1 allows a limited amount of timing error in the data that causes a trigger action to execute, and Level 0 guarantees the convergence of the memory nodes of the discrimination network for a trigger.

We developed techniques that efficiently implement each consistency level. They include the Stability-Lookaside Buffer (SLB) for memory node stabilization, the shadow table for virtual α nodes, the Concurrent Token Set (CTS) detection and token processing architecture, and the Duplicate-Lookaside Buffer (DLB) for the prevention of duplicate compound tokens. These techniques will improve the overall performance of the system and allow the users to choose desired semantic consistency levels for their triggers.

CHAPTER 1
INTRODUCTION

Active database systems [20],[22],[24],[25],[59],[70],[73] are able to respond automatically to the situations of interest that arise in the data. The behavior of an active database is described using triggers [74]. Many database products provide synchronous triggers where trigger condition testing and action execution are always done as part of the update transaction [43],[45],[71]. The problem with synchronous triggers is that they can cause update response time to be slower.

TriggerMan [6],[9],[13],[16],[36],[56],[57] is an asynchronous trigger processing software system that checks trigger conditions and runs trigger actions outside of transactions that update the data sources. In other words, we are assuming that the trigger processing system is separate from the data sources. The data sources can be either database tables or generic data sources. TriggerMan allows triggers to have a condition based on multiple data sources (database tables). In an asynchronous trigger system, a discrimination network is used to check the condition of a trigger. TriggerMan is designed to use Gator networks, and take advantage of their good performance properties for testing join trigger conditions.

A discrimination network consists of condition testing nodes, memory nodes (α nodes, β nodes), and a P-node. A P-node is the root node of a discrimination network. The insertion of a tuple into a P-node means that a combination of tuples satisfying the

trigger condition has been found, and the trigger action needs to be executed. The changes to data sources are delivered to an asynchronous trigger system in the form of a *token*. The term *token* was coined in the research on AI production systems [1],[2],[3], [12],[28],[29],[30],[50]. Memory nodes of a discrimination network are updated using tokens. Another way of updating memory nodes is to refresh them periodically as proposed by Adiba [5].

A *view* [8],[62],[67] is a derived table defined in terms of base (stored) tables. Thus, a view defines a function from a set of base tables to a derived table. This function is typically recomputed in whole or in part every time the view is referenced using a procedure called query modification [63]. A view can also be materialized to provide fast access to data by storing the tuples of the view in the database [32]. The maintenance of a materialized view can be done in a synchronous way. However, synchronous view maintenance imposes a significant overhead on update transactions and cannot be tolerated by many applications.

Materialized view maintenance [7],[51],[69] has been studied widely, especially in new applications such as data warehousing [14],[18],[32],[42],[66],[76]. View maintenance in a warehousing environment is inherently asynchronous [18],[52],[76]. There are many similarities between the materialized views and the contents of memory nodes and P-nodes of a discrimination network (see Section 2.1). We believe that discrimination networks could be used in maintaining materialized views. Therefore, the TriggerMan system can maintain materialized views asynchronously with ease and efficiency.

This research consists of two parts. In the first part, we introduced techniques to improve the performance of TriggerMan such that the techniques do not compromise semantic correctness of trigger processing. The techniques include *Gator* network dynamic restructuring, efficient processing of a large token set, parallel resource utilization, and parallel processing of data and trigger condition and action.

To find an optimized Gator network structure, we collect or estimate various statistics about the data sources and the cost of conditions of the trigger that is being defined. However, those statistics can either be inaccurate or change over time. Therefore, it is necessary to accumulate the statistics and restructure the Gator networks accordingly during run time, while keeping the overhead of dynamic restructuring at a minimum. Gator network dynamic restructuring is covered in Chapter 3.

Assume a set of tokens is arriving at the same node *n* simultaneously. If the size of the token set were large, then it would be more efficient to process it using a so-called *set query approach*. This approach stores the tokens in a temporary table and joins the temporary table with the sibling nodes to find the compound tokens and propagate them to the parent node of *n*. However, if the size of the token set were small, then the joining with the sibling nodes token by token would be more efficient. This method is called a *tuple query approach*. The determination of the crossover point between the large sets and the small sets is covered in Chapter 4.

Parallel resources include processors, processes, main memory, and disks. To utilize a given hardware and software (database system) environment, we need to estimate the amount of the parallel resources required for each Gator network node. The

result will be used in requesting parallel resources before processing the SQL queries for the Gator network node. Parallel resource utilization is covered in Chapter 5.

Processing tasks in parallel will increase the scalability of the system. The tasks include trigger condition checking, trigger action execution, and data (memory nodes and base tables) processing. A discussion about the parallel processing of these task types appears in Chapter 6. However, parallel token processing causes problems in the semantic correctness of trigger processing.

The second part of the research focuses on incorporating parallel token processing into an asynchronous trigger system in a productive way. To accomplish this, we define four *consistency level*s in trigger processing. The purpose of *consistency level* is to achieve maximum performance while allowing acceptable and anticipated problems. The lower the level is, the higher the performance is. Four consistency levels can be summarized as follows: Level 3 provides serial token processing semantics for a trigger, Level 2 executes trigger actions using all and only correct data, Level 1 allows a limited amount of *timing error* in the data that executes a trigger action, and Level 0 guarantees the convergence of the memory nodes of the discrimination network for a trigger. Each level is covered by a specific section of Chapter 7 as follows.

Level 3 is the highest consistency level of trigger processing. Level 3 consistency for a trigger, *T*, can be achieved by serially processing the tokens that arrive at the discrimination network for *T*. Hence, Level 3 provides the lowest performance among the four consistency levels. When a virtual α node is in the discrimination network for a trigger, to provide Level 3 consistency for the trigger, a special table called a *shadow table* is needed for the virtual α node. The whole technique for Level 3 consistency and a

technique that resolves the so-called *duplicate compound token* problem are presented in Section 7.3.

Level 2 consistency allows the out-of-order execution of multiple instantiations of the action of a single trigger. It is known that some tokens that arrive at the same discrimination network can be processed in parallel and provide Level 2 consistency. Such a group of tokens is called a *Concurrent Token Set* (CTS). The technique for CTS detection and a proposed architecture for efficient token processing are introduced in Section 7.4.

Since Level 1 allows a limited amount of *timing error*, all the tokens that arrive at a trigger system within a limited time interval can be processed in parallel. However, simple concurrent token processing can corrupt the memory nodes of a discrimination network and break one of the conditions for Level 1 consistency. Therefore, a mechanism that guarantees the convergence of memory nodes is needed. We developed the *Stability-Lookaside Buffer* (SLB) for the stabilization of memory nodes. Our proposed strategies that use the SLB to stabilize α nodes and β nodes are presented in Section 7.5.

Although in Level 0 consistency the timing error is allowed to extend indefinitely, the memory nodes of a discrimination network are required to converge. The stabilization of a stored α node and a β node that does not have virtual α node descendents can be achieved using the strategies introduced in Section 7.5. However, a strategy for the stabilization of a β node that has virtual α node descendents needs to be developed. Our proposed strategy for such a β node and a proof of the stabilization of the node are presented in Section 7.6.

To help system developers, a discussion about the implementation alternatives of an asynchronous trigger system is given in Section 7.7. Finally, a summary of this research appears in Chapter 8.

CHAPTER 2
BACKGROUND

Active database systems are able to respond automatically to situations of interest that arise in the databases. The behavior of an active database is described using triggers. For the efficient trigger condition checking, a trigger processing system can use *discrimination networks*. *TriggerMan* is an asynchronous trigger processing system that checks trigger conditions and runs trigger actions outside of a DBMS or other data sources.

This chapter is organized as follows: Section 2.1 compares three kinds of *discrimination networks*, Section 2.2 explains how tokens are propagated in a discrimination network, and Section 2.3 briefly describes the *TriggerMan* system.

2.1 Discrimination Networks

Discrimination networks have tree structures. The root of a discrimination network is usually drawn at the bottom. Rete, TREAT, and Gator networks are major discrimination networks and their prominent difference is in the shape of the tree structure. A discrimination network has (selection and/or join) condition checking nodes, memory nodes, and a P-node.

There are two kinds of memory nodes: the α memory nodes (simply, α nodes) and the β memory nodes (simply, β nodes). An α node holds the result of applying a

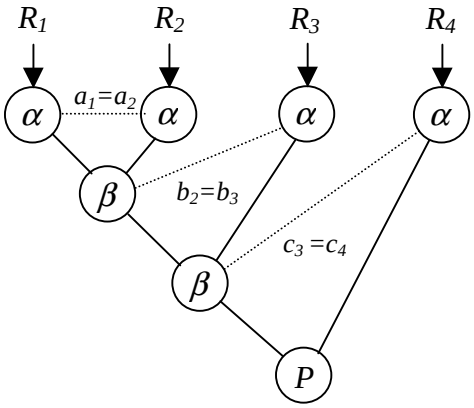
selection condition to a database table. A β node holds the result of the join of other memory nodes.

An α node can be either a *stored* α node or a *virtual* α node. An α node that contains the qualifying data in it is called a *stored* α node. An α node that contains the predicate describing the contents of the node rather than the qualifying data is called a *virtual* α node. Each β node is a stored β node. A stored memory node and a virtual memory node are analogous to a materialized view and a real view, respectively. In Rete terminology, the root node of a discrimination network is called the P-node.

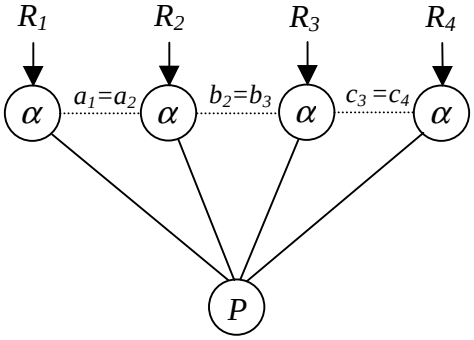
The Rete algorithm was proposed by Forgy [23] to do the pattern matching for rules in OPS5 [11]. The Rete algorithm uses a tree-structured sorting network for the productions. We call the network a *Rete network*. A Rete network has both α memory nodes and β memory nodes. Every α node in a Rete network is a stored α node. Each β node in a Rete network has exactly two inputs (children). A possible Rete network for the rule *a_dummy_rule* (Figure 2.1) is shown in Figure 2.2 (a). Note the network has a binary tree structure.

define rule	*a_dummy_rule*
if	$R_1.a_1 = R_2.a_2$ **and** $R_2.b_2 = R_3.b_3$ **and** $R_3.c_3 = R_4.c_4$
then	action

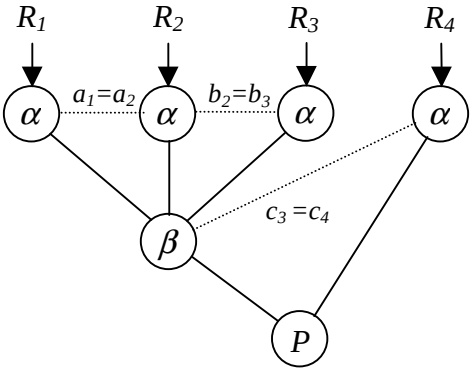
Figure 2.1: A dummy rule



(a) A Rete network



(b) A TREAT network



(c) A Gator network

Figure 2.2: Various types of discrimination networks

The TREAT match algorithm was developed for AI production systems by Miranker [60],[61],[72]. A TREAT network has only α nodes as the memory nodes. As a result, TREAT removes the overhead of maintaining β nodes. In a TREAT network, the order in which α nodes are joined can be recomputed dynamically. Every α node of a TREAT network is a stored α node. The TREAT network for the rule *a_dummy_rule* (Figure 2.1) is shown in Figure 2.2 (b). To reduce the storage requirement, Hanson developed the A−TREAT algorithm [33]. In an A−TREAT network, an α node can be either a stored α node or a virtual α node. Note the height of the tree is only one.

The Gator (Generalized TREAT/Rete) network is developed for active database rule systems and production system interpreters by Hanson [34],[37]. A Gator network has general tree structure and shows good performance but with added system complexity [34]. Rete networks and TREAT networks can be seen as the subsets of Gator networks. A Gator network has both α nodes and β nodes as the memory nodes. An α node in a Gator network can be either a stored α node or a virtual α node. A possible Gator network for the rule *a_dummy_rule* (Figure 2.1) is shown in Figure 2.2 (c). Note the network has a general tree structure.

Performance studies in [72] indicate that, in a database environment, TREAT usually outperforms Rete, but Rete is better than TREAT in a few cases where the frequency distribution of updates to different relations in the rule condition is skewed. Gator networks, if properly tuned, have the potential to perform well in all cases [9],[56].

2.2 Token Propagation in a Discrimination Network

In an asynchronous trigger processing system, as an implementation method, materialized memory nodes are stored in a commercial database (host DBMS). When one or more tokens arrives at a node, the *query modification technique* [36] is used to propagate them. To make the *query modification technique* work, a *tuple query template* (TQT) and a *set query template* (SQT) need to be created for each memory node. They are created based on the discrimination network structure and the trigger condition. A TQT and an SQT are stored in each memory node.

When a + token arrives at a memory node, *n*, the TQT stored in *n* is modified using the token. When many + tokens arrive at *n* simultaneously, the tokens are stored into a temporary table and the SQT stored in *n* is modified using the temporary table. The modified TQT or SQT is submitted to the host DBMS for execution. If the query has a result, then the tuples in the result are propagated to the parent node of *n*. If the query has no result, then the processing of the token(s) stops. A detailed explanation appears in [36].

An example is given in Figure 2.3. In the figure three table schemas and a trigger, *enroll_DBMS*, are defined. A Gator network for *enroll_DBMS* is also shown in the figure. In the Gator network, the α node α_1 logically contains the result of the following query:

select * **from** *class* **where** *class.cname* = ”DBMS”

Similarly, the β node β_1 logically contains the result of the following query:

select * **from** *class, enroll* **where** *class.cname* = ”DBMS” **and** *class.cno* = *enroll.cno*

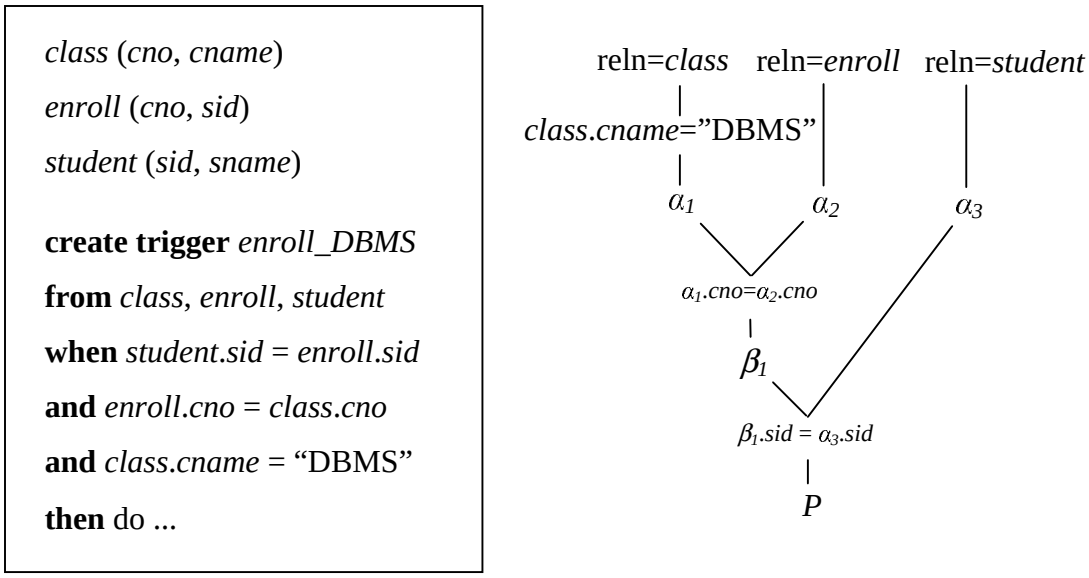


Figure 2.3: A trigger and a Gator network for it

Suppose a token, *t*, is inserted into α_2. Then the system modifies the TQT stored in α_2 using *t*, and creates the following query:

select α_1.*, *t*.* **from** α_1 **where** α_1.*cno* = *t.cno*

Here, *t*.* is expanded into a list of one or more constants (all the fields of *t*), and *t.cno* represents the value of *cno* field of *t*. The above query is then passed to the query processor for execution. Each tuple in the result is transformed into a + compound token and then propagated to β_1, the parent node of α_2.

If many tokens are inserted into β_1 at the same time, they are stored into a temporary table, *temp*. The system modifies the SQT stored in β_1 using *temp*, and creates the following query:

select *temp*.*, α_3.* **from** *temp*, α_3 **where** *temp.sid* = α_3.*sid*

The above query is then passed to the query processor for execution. Each tuple in the result is transformed into a + compound token and then propagated to *P*, the parent node of β_1.

The propagation of a – token can be done in two forms -- an atomic token or a compound token. Subsections 7.5.1, 7.5.2, and 7.6.3 explain – token propagation in detail.

2.3 TriggerMan -- Asynchronous Trigger Processing System

TriggerMan basically contains the main components of an active database system, but is separated from any specific database. However, TriggerMan is implemented on a specific database to store its catalog and state information. TriggerMan receives update descriptors from the data sources (databases) asynchronously, and allows the update transactions on the databases to execute at an uninterrupted speed. A TriggerMan client can define a trigger on multiple data sources and can register for the events provided by the TriggerMan server. The TriggerMan system consists of the following components:

- The server, which lives inside of a commercial DBMS (in current implementation, Informix Dynamic Server with Universal Data Option [46], we call this Informix/UDO), is a passive module with a set of user defined routines (Informix/UDO terminology).
- Data source applications are programs that transmit a sequence of update descriptors (tokens) to the server describing updates that have occurred in the data sources.

- Client applications create triggers, drop triggers, register for events, receive event notifications when triggers fire, etc.
- The driver is a program that periodically invokes a special function, TmanTest(), in the server, allowing trigger condition checking and action execution to be performed; more than one instance of the driver can run at the same time to fulfill the performance requirement.
- The console is a special client application program that lets a user directly interact with the system to create or drop triggers, start or shut down the system, etc.

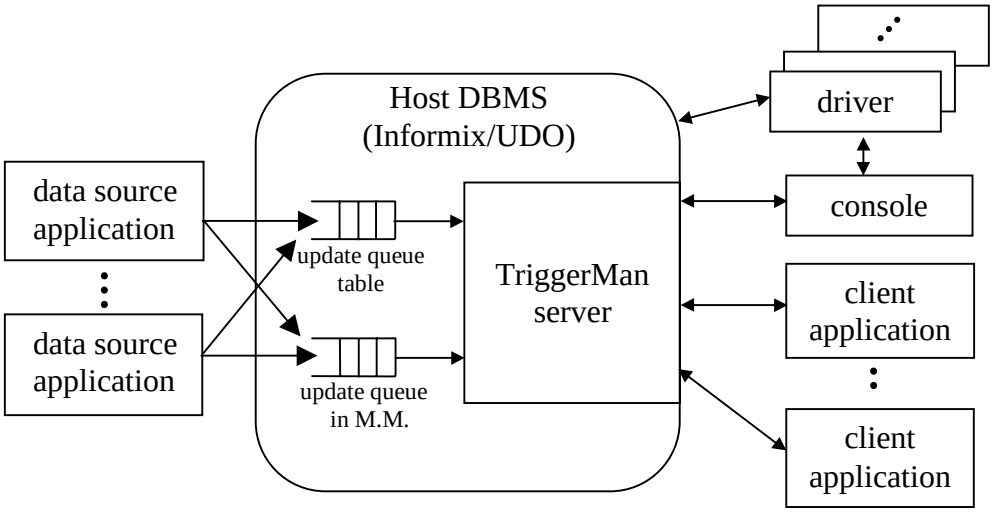


Figure 2.4: The architecture of the TriggerMan system

The general architecture of the TriggerMan system is illustrated in Figure 2.4. Having more than one driver program concurrently invoking the TmanTest() function can allow higher throughput for TriggerMan by letting concurrent trigger processing

activities take place inside the host DBMS server. The TriggerMan system catalog contains data source and trigger definitions, structures and statistics of discrimination network nodes, etc. The catalog and the persistent update queues will be stored in database tables of the host DBMS.

Two libraries that come with TriggerMan allow writing of client applications and data source applications. These libraries define the client application programming interface (API) and the 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 applications can be written using the data source API [13]. Examples of data source programs are generic data sources that send streams of update descriptors to TriggerMan, and DBMS gateway programs that gather updates from the DBMSs and send them to TriggerMan.

As Figure 2.4 shows, data source applications can either place update descriptors in a persistent queue, or in a volatile queue in shared memory. A persistent update queue is an ordinary host DBMS table created and used by TriggerMan. A volatile queue in shared memory is used to hold update descriptors that do not need to be recovered in case of a system failure. It is the duty of the application designer to determine if update descriptors sent to TriggerMan need to be recoverable.

CHAPTER 3
GATOR NETWORK DYNAMIC RESTRUCTURING

In the TriggerMan system, when a trigger is defined, an optimized Gator network is produced to test the condition of the trigger (the same thing happens with a materialized view). An optimized Gator network for a trigger condition is produced by our optimizer [9],[56]. The optimization of a Gator network depends on many factors like the statistics on the data sources and the costs of the selection and/or join conditions in the condition of the trigger that is being defined.

Some of these factors are estimated since they are unavailable at the time of optimization. They are:

- Database relation (data source) update frequencies
- Selectivity factors of the selection conditions associated with α nodes
- Join selectivity factors (JSFs) of the join conditions between two nodes

Due to the statistical assumptions made in the estimation of these values, errors are inevitable [17]. Furthermore, since intermediate estimations can be operands of further estimations, inaccuracies in current estimates propagate and aggravate later ones [47].

Other factors, like the relative token insertion frequencies (the sum of all the relative token insertion frequencies of α nodes of a Gator network is 100) and the sizes of

the memory nodes, change over time [56]. Let us consider a simple Gator network with three α nodes in Figure 3.1.

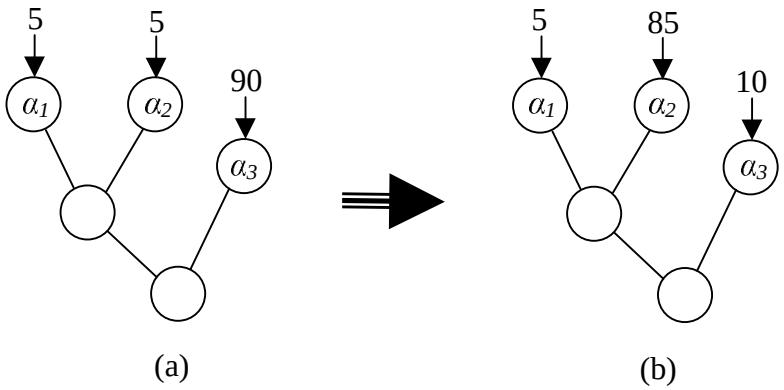


Figure 3.1: Token insertion frequencies

In the figure, (a) is the initial Gator network. The tops of incoming arrows represent the relative token insertion frequencies (5, 5, and 90) that were estimated when the network was built. However, we can assume that the values of these frequencies will change to 5, 85, and 10, respectively, over time (see (b) of Figure 3.1). Let us temporarily ignore other factors influencing the Gator network structure. This will make it easier to understand the effect of the changes in the relative token insertion frequencies. It is certain that the initial network structure (shown in (a) of Figure 3.1) is no longer optimal in propagating the incoming tokens with the new frequencies. If that is the case, we need to restructure the Gator network during run time.

Geier [26] made a similar argument for computer network reengineering (restructuring). The factors that he considered in modifying the computer network include new applications, technology shifts, and organizational resizing. Typically, the

cycle of computer network reengineering is much longer than that of Gator network restructuring. In doing network reengineering, he did not consider the cost of analyzing the modification feasibility. This is one of the differences between the computer network reengineering and Gator network restructuring.

Bodorik, Riordon, and Pyra made an effort to overcome the inaccuracies in the initial estimations for determining query processing strategies in distributed databases [10]. In their study, the minimization of the delay of the distributed query execution was crucial, because the correction of the query processing strategy was performed while the query was being processed. Hence, they adopted a computationally simple decision method and prepared alternative heuristic strategies in a background mode. However, optimal Gator network restructuring is an expensive operation involving many factors, including new β node construction. Therefore, we could not prepare alternative plans in advance.

Gator network restructuring is surprisingly similar to the equipment replacement in operations research [68]. We believe that we can get an idea of what the Gator network restructuring system should look like by comparing the two systems. Section 3.1 compares the two systems. Section 3.2 discusses test schedule for a Gator network restructuring. Section 3.3 introduces a formula that can be used in testing the Gator network for restructuring. Section 3.4 gives two examples of Gator network restructuring. Finally, Section 3.5 summarizes this chapter.

3.1 Gator Network Restructuring and Replacement Theory

First, let us introduce replacement theory in operations research [68].

Replacement theory was originated from the observation that equipment deteriorates with age; that is, the longer the equipment is retained, the higher the cost of operating it. Thus, as an alternative, it may be profitable to acquire new equipment that is more economical to operate. The fundamental problem that one is faced with is to make an appropriate balance between the cost of increased upkeep of the old equipment and the acquisition cost and reduced upkeep of new equipment.

Table 3.1 shows a comparison between Gator network restructuring and the equipment replacement in operations research. Intermittently operating equipment is characterized by the fact that its operation depends on the user's request; thus, the equipment ages only when it provides service. The service that is provided by a Gator network is the token propagation through the network. The tokens arriving at the Gator network change the statistics decreasing the optimality of the Gator network. Therefore, a Gator network is a kind of an intermittently operating equipment, because it ages (Gator network becomes sub-optimal) when it propagates tokens through the Gator network (in other words, when it operates).

Some parameters change with time after the latest optimization. Hence, the current Gator network structure becomes sub-optimal against the current statistics. In general, we can say that the performance of a Gator network deteriorates over time. Therefore, we need to restructure Gator networks during runtime. Since the rate of deterioration is hard to predict, we need to test the performance of the Gator network and restructure it depending on the test result.

Table 3.1: Gator network restructuring and equipment replacement

FACTORS	Gator network restructuring	Equipment replacement
Basic assumption	- Gator network performance deteriorates with age[a]	- equipment deteriorates with age
Alternative plan	- restructure Gator network	- acquire new equipment
Input parameters	- restructuring cost[b] - increased performance of the new Gator network	- acquisition cost - reduced upkeep of the new equipment
Decision variable	- restructuring decision	- replacement age
Distinctive features	- simple comparison to get restructuring decision[c] - costs of optimality test and statistics gathering does not affect replacement decision[d]	- cost estimation curve to get optimum replacement age - replacement decision cost is not accounted

[a]As factors change over time, a Gator network becomes sub-optimal and deteriorates with age.
[b]Restructuring involves re-optimizing a Gator network and priming newly created memory nodes.
[c]The future trends of the costs of the triggers are not projected.
[d]The costs are considered as the necessary evil or tax.

The restructuring cost includes not only the re-optimizing cost of the Gator network, but also the cost of priming new memory nodes for the new Gator network. Priming is the process in which the nodes of the discrimination network are loaded with the tuples that match the selection/join conditions associated with these nodes. Restructuring cost is also known as *preparation time* in Section 3.2. Because thousands of triggers will be defined in TriggerMan, projecting the future trends of the costs of

thousands of Gator network would be expensive. Therefore, we are comparing the benefit of restructuring with the cost of it to decide the restructuring.

We are going to accumulate node statistics and selection/join selectivity factors of the Gator networks while TriggerMan is in operation. Using the accumulated statistics the performance of a Gator network will be tested on a schedule that is specific to the Gator network. A Gator network will be restructured depending on the result of the test. We will not include the costs of statistics gathering and optimality testing into the restructuring cost of the new Gator network. We will consider the costs as the necessary evil or tax. However, we are trying to reduce the optimality testing cost by increasing the test schedule when it is appropriate.

3.2 Test Schedule for Restructuring

After the optimizer builds the Gator network, in order to make it operational we need to prime the Gator network (load its stored α nodes and β nodes with data). Optimizing a Gator network and priming the optimized Gator network are time-consuming operations. Therefore, it is inefficient to restructure more often than is necessary. Let ***preparation time*** be the optimization time plus the priming time of a Gator network. Because each Gator network has a different change rate of statistics, the test schedule for a restructuring needs to be different for each trigger.

We considered using the difference between the estimated statistics and the observed statistics of a Gator network as an indicator of the sub-optimality of the Gator network. This was done by Kabra [55] who detected the sub-optimality of a query execution plan during query execution. Obviously, sub-optimality detection that is based

on the statistics change is cheaper than one based on the comparison of Gator network costs obtained using the optimizer. However, we will use the latter based on the following observations:

- It is hard to see a simple change in statistics definitely indicates the sub-optimality of a Gator network since Gator network optimization is more complex than query optimization [9].
- The cost of optimization is justified by the following facts; the restructuring of a Gator network is less frequently done than the re-optimization of a query and Gator network restructuring involves priming cost (which is more expensive than the optimization cost) when restructuring is done.
- The Gator network optimizer considers the change in statistics when deciding sub-optimality of a Gator network and gives exact indication.

In replacement theory, the equipment service age is defined as the total number of registered operational units of the equipment since acquisition. For example, in the case of a vehicle, the operational units would be miles. The age would then be the number of miles registered by the vehicle odometer. We believe that a Gator network needs to provide service without being tested for restructuring for the time period proportional to its preparation time; otherwise, too much work will be wasted in testing the optimality of the Gator networks. That is, the restructuring schedule of Gator network should be based on the operation time, because it is a kind of an intermittently operating equipment. Therefore, we have included the preparation time among the factors that determine the test schedule for restructuring.

We will also use a system-wide constant in calculating the schedule to allow the TriggerMan system administrator to control the initial test schedule of all Gator networks in the system. Because of the different change rates of statistics on Gator networks, we will assign and maintain a factor, *f*, for each Gator network so that it can control the test schedule of the Gator network individually.

In summary, the components that control the test schedule of a Gator network are:

- π : preparation time of a Gator network, in seconds
- C : system-wide constant controlling the schedules of all Gator networks ($\geq$ 1.0)
- f : factor controlling the restructuring schedule of a Gator network, $f = 2^n$ (n: integer, $n \geq 0$)

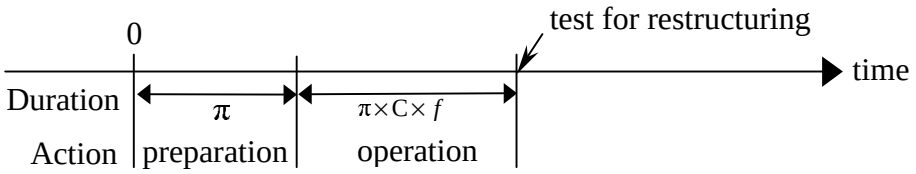


Figure 3.2: Gator network restructuring schedule

Once every $\pi \times C \times f$ seconds of operation we will test the optimality of a Gator network against the recent statistics gathered during the operation (see Figure 3.2). Then, we will restructure it conditionally based on the result (details are explained in Section 3.3). The π in $\pi \times C \times f$ formula is the preparation time of the most recent preparation. Constant C can be set to 1 initially and can be adjusted during later performance tuning.

The initial value of factor *f* of each Gator network will be 1. The value changes depending on the result of the restructuring test for the Gator network (details are explained in Section 3.3).

3.3 Restructuring Test of Gator Network

If the Gator network has operated for $\pi \times C \times f$ seconds since the recent restructuring, then check the necessity for restructuring using the statistics accumulated since the recent restructuring. The guidelines for the Gator network restructuring are:

- The restructuring decision should be based on the cost and the benefit of the restructuring (refer to *distinctive features* factor in Table 3.1); if the benefits of restructuring is greater than the costs of it, then restructure the Gator network (refer to Figure 3.3).
- The cost of testing should be as little as possible to reduce overhead (Refer to *distinctive features* factors in Table 3.1).
- Depending on the restructuring decision made, the next schedule needs to be changed. Otherwise, there will be too much optimality testing overhead for the Gator network that follows the estimated statistics or the benefit of opportune restructuring cannot be reaped (refer to the factor *f* in Section 3.2).

Figure 3.3 illustrates how to analyze the feasibility of restructuring the Gator network. The decision to restructure the Gator network is based on the result of the comparison of the costs and the benefits. If the benefits exceed all costs involved, then the decision should be to incorporate the restructuring.

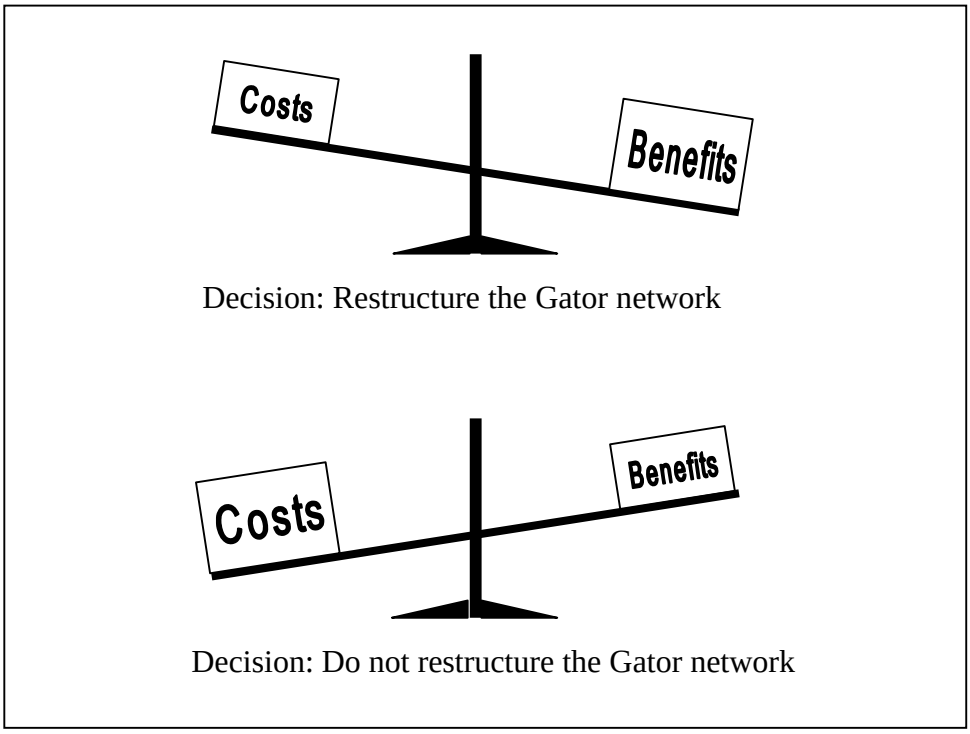


Figure 3.3: Analyzing restructuring feasibility

Keeping the above guidelines in mind, see Figure 3.4 where the performance axis specifies the ratio of the observed performance to the optimal performance $\left(\frac{\text{performance}_{\text{observed}}}{\text{performance}_{\text{optimal}}}\right)$. In Figure 3.4, the dotted lines passing through the time axis and the actions in parenthesis are the future events. After Σ seconds of operation, a Gator network is tested for a restructuring by comparing the benefit of restructuring and the cost of it. The benefits and the costs of the restructuring can be obtained based on two assumptions.

The *first assumption is that the difference between the estimated statistics and the observed statistics of the Gator network will maintain, in the future, at least for the same amount of time as the time during which the statistics were gathered*. Therefore, the

average performance (p' in Figure 3.4, $0 < p' \leq 1.0$) of the Gator network will hold or decrease in the future for the time during which the statistics were accumulated (Σ seconds in Figure 3.4). A similar forecasting method is used successfully in the LRU page replacement policy for the virtual memory in the computer operating systems and in the focus forecasting method [15].

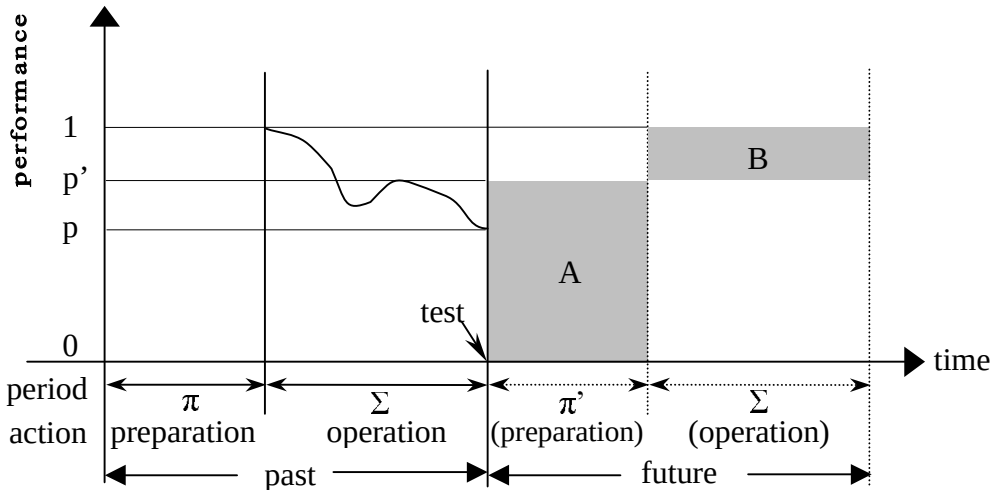


Figure 3.4: Testing a Gator network for restructuring

The *second assumption is that the time of the next preparation will be the same as the time of the previous preparation*. The same assumption is held in the simplest type of replacement situations in replacement theory. Therefore, π' will be equal to π in Figure 3.4.

The benefit of restructuring will be the guaranteed increment of throughput of the new Gator network. In obtaining the throughput increment of a Gator network, we multiply the performance increment by the time period during which the increased

performance will be maintained. By the first assumption given above, this time period will be the same as the past operation time (Σ seconds in Figure 3.4). Now we have the benefit (the area of the shaded region B in Figure 3.4):

$$(1 - p') \times \Sigma \tag{3.1}$$

Because the normal operation cannot be performed during the preparation of the new Gator network, the cost of restructuring can be obtained by multiplying the average performance (performance of the old Gator network) by the preparation time (π seconds in Figure 3.4). Now we have the cost (the area of the shaded region A in Figure 3.4):

$$p' \times \pi \tag{3.2}$$

The average performance (p') during a period can be calculated using the statistics accumulated during that period. Since the performance is directly related to the cost (the concept used by the cost-based optimizer), p' can be obtained from the cost of the current Gator network (G_C) against the accumulated statistics.

We actually need two costs to get p': the cost of G_C (C_C) and the cost of the optimal Gator network (G_{OPT}). The calculation of C_C is straightforward; pass G_C to the cost calculation module of the optimizer and let the module calculate C_C using the accumulated statistics. To obtain the cost of G_{OPT} (C_{OPT}), we need to invoke the Gator network optimizer and find G_{OPT}. When we invoke the optimizer we will use the current Gator network as the initial starting state (Our optimizer uses iterative improvement [9] as its main algorithm and iterative improvement needs many starting states). This is based on the heuristics that the optimal Gator network will be similar to the current Gator network. Using the current Gator network as the initial starting state guarantees that the

optimizer will find a Gator network whose cost is as low as the current one. We will use the cost of the network found by the optimizer as C_{OPT}. It is expected that the portion of priming time in the preparation time is far greater than that of the Gator network optimization time. Therefore, the execution cost of the optimizer in making the restructuring decision is justified.

Since the performance of the Gator network is inversely proportional to its cost, we have

$$1 : p' = \frac{1}{C_{OPT}} : \frac{1}{C_C}$$

Hence,

$$p' = \frac{C_{OPT}}{C_C} \tag{3.3}$$

Substituting (3.3) in the benefit formula (3.1) gives

$$\left(1 - \frac{C_{OPT}}{C_C}\right) \times \Sigma \tag{3.4}$$

Substituting (3.3) in the cost formula (3.2) gives

$$\frac{C_{OPT}}{C_C} \times \pi \tag{3.5}$$

Using formulas (3.4) and (3.5) we can formulate the restructuring condition:

$$\left(1 - \frac{C_{OPT}}{C_C}\right) \times \Sigma \geq \frac{C_{OPT}}{C_C} \times \pi \tag{3.6}$$

Based on the analysis in Figure 3.3, if (3.6) is true, we will restructure the Gator network. Following the last guideline shown at the beginning of Section 3.3, if the restructuring is to be done, we will decrease *f* by half. Otherwise (no restructuring), we

will double the value of *f*. By doing this, we can reduce the costs of the optimality test of the Gator network.

3.4 Gator Network Optimality Testing Examples

In this section we are presenting two examples of Gator network optimality testing for dynamic restructuring. As common conditions for the two examples, let us assume that π (preparation time) is 1200 seconds, C is 1, C_C is 140, and C_{OPT} is 90.

In the first example, we have the additional conditions of *f* equal to 1.0 and Σ equal to 1200 seconds. Because Σ (1200 seconds) is equal to $\pi \times C \times f$ ($1200 \times 1 \times 1.0 = 1200$ seconds) (see Figure 3.2), then it is time to test the performance of the Gator network and determine the restructuring of it.

The value of formula (3.6) is

$$\left(1 - \frac{C_{OPT}}{C_C}\right) \times \Sigma \geq \frac{C_{OPT}}{C_C} \times \pi = \left(1 - \frac{90}{140}\right) \times 1200 \geq \frac{90}{140} \times 1200$$

$$= 5 \geq 9$$

$$= \text{false}$$

Hence, restructuring is not to be done, and *f* is doubled becoming 2.0.

The second example is a continuation of the first example. Now we have *f* equal to 2.0 and Σ equal to 2400 seconds as additional conditions. Because Σ (2400 seconds) is equal to $\pi \times C \times f$ ($1200 \times 1 \times 2.0 = 2400$ seconds) (see Figure 3.2), then it is time to test the necessity of a restructuring. The value of formula (3.6) is

$$\left(1 - \frac{C_{OPT}}{C_C}\right) \times \Sigma \geq \frac{C_{OPT}}{C_C} \times \pi = \left(1 - \frac{90}{140}\right) \times 2400 \geq \frac{90}{140} \times 1200$$

$$= 10 \geq 9$$

= true

Therefore, the Gator network will be restructured, and *f* will be reduced by half leaving 1.0.

3.5 Summary

To decide whether to restructure a Gator network or not, the performance (optimality) of the Gator network needs to be evaluated. We reuse the cost function of the Gator network optimizer in evaluating the performance of a given Gator network. Since the optimality test has its own cost, it should not be done more often than necessary. To reduce the cost of the optimality test, a formula that contains factors (the *preparation time* and the *adjusting factor*) specific to each Gator network is developed. The value of the formula determines when to test the optimality of a Gator network.

The decision to restructure is based on the cost and the benefit of restructuring. The cost of restructuring of a Gator network is obtained by multiplying the average performance and the preparation time of the current Gator network, since the normal operation cannot be performed during the preparation of the new Gator network. The benefit of restructuring is the increased throughput obtained using the new Gator network. Therefore, the benefit of restructuring of a Gator network was obtained by multiplying the performance increment and the time of operation after the previous restructuring of the Gator network. An expression whose value determines whether to restructure a Gator network is developed.

CHAPTER 4
OPTIMAL PROCESSING OF A TOKEN SET

When a token arrives at a memory node (α or β node), the TriggerMan system will modify the query that is stored in the node (see Section 2.2 for more details). The modified query will be submitted to the host DBMS to find tuples that match the token. The matching tuples will be used in forming compound tokens that will be sent to the parent node of the memory node. We can imagine the situation where multiple + tokens arrive simultaneously at a β node. If we process the tokens coming from a data source concurrently and accumulate the tokens from that data source arriving at an α node, then an α node can also have multiple tokens arriving simultaneously.

There are two ways to propagate the tokens that arrive simultaneously at a memory node. One way is to form a query for each token by modifying the tuple query template (TQT, see Section 2.2) and executing the queries. The join degree of each query will be *d* when the number of siblings of the memory node is *d*. The number of queries to execute will be *m* given that the number of tokens is *m*. We will call this the *tuple query approach*. The other way is to transform the token set into a temporary table, form a query by modifying the set query template (SQT, see Section 2.2), and execute the query. The join degree of the query will be *d*+1 given that the number of siblings of the memory node is *d*. We will call this the *set query approach*.

When a database system receives a query, it goes through the syntax-checking phase and the query optimization phase [63]. The query optimization time increases rapidly as the number of variables in the query increases, because the number of possible plans increases combinatorially [19],[53]. It is evident that the smaller the set of + tokens is, the more efficient the tuple query approach is; similarly, the larger the set of + tokens is, the more efficient the set query approach is. Finding the crossover point between the large sets and the small sets is an optimization problem and will be studied in this chapter. We will estimate the costs of the queries using the optimizer provided by the host DBMS and apply the interpolation on the result to determine the crossover point of each Gator network node (more precisely, of every <*n*, parent(*n*)> pair of each memory node *n*).

This Chapter is organized as follows: Section 4.1 discusses how to find the crossover point, Section 4.2 discusses the method of estimating the costs of the queries, Section 4.3 introduces the process of deciding the query approach type, Section 4.4 presents the algorithms that propagate token sets, and Section 4.5 presents the summary and further studies.

4.1 Finding Crossover Point

A crossover point is a number such that if the number of tokens in a set that arrive at a node exceeds the number, then it is more efficient to take the set query approach to propagate the token set. Otherwise, taking the tuple query approach is more efficient. Because the complexity of a query determines its optimization time and optimization time determines the crossover point, each < *n*, parent(*n*)> pair of each node *n* needs to

have a unique crossover point. To make the explanation simple, we assume that each memory node has a single parent node.

If a + token arrives at a node, then the tuple query approach will probably be beneficial. Let us assume that a set of two tokens arrived at a node *n*. Because of the increased optimization time, the cost of joining the set of two tokens with the siblings of *n* would probably be greater than twice the cost of a tuple query. Let the token set size be *k*. As *k* increases, the cost of the set query could be lower than *k* times the cost of a tuple query. This is due to the linear increase of the cost of the tuple query approach.

The determination of the crossover point of a node *n* is possible based on the assumption that we are making; the cost of joining a temporary relation (TR) with the siblings of *n* is linearly proportional to the number of tuples in TR. To determine a crossover point of *n*, three kinds of costs are needed (see Figure 4.1):

- C_1: the cost of preparation and execution of the query joining a token arriving at *n* with the siblings of *n* (the nodes that have the same parent as *n*)
- C_2: the cost of preparation and execution of the query joining the temporary table of size 2 with the siblings of *n*
- C_k: the cost of preparation and execution of the query joining the temporary table of size k with the siblings of *n*

We cannot say that C_i increases perfectly linearly with the increase of *i*. If we choose *k* as close as the real crossover point, then the error of estimating the crossover point will be minimized (details are omitted). However, we do not know the crossover point in advance. Therefore, at first, some number that is substantially larger than 2 can

be used as *k*, and then, after we have some idea about the real crossover point, we can choose a universal *k* value.

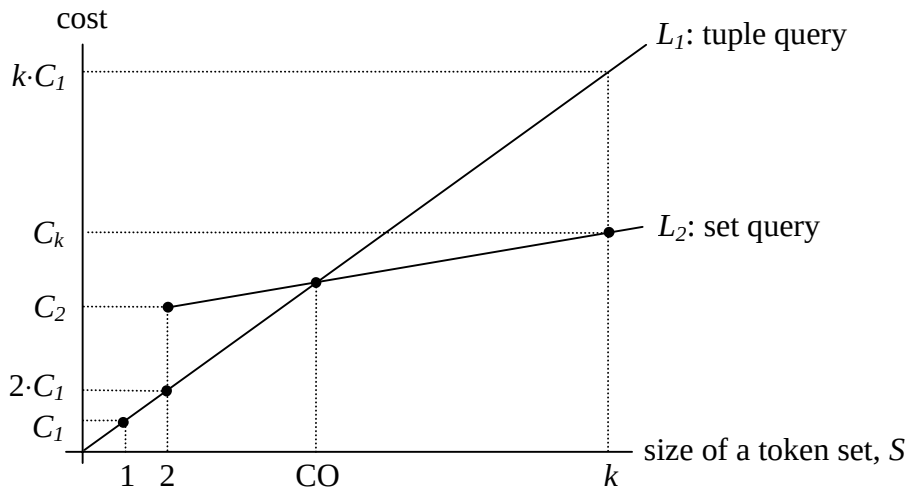


Figure 4.1: Cost of the tuple query and the set query

Let *S* be the set of + tokens that arrives at a memory node *n*. In Figure 4.1, lines L_1 and L_2 denote the costs of the tuple query approach and the set query approach, respectively, in propagating *S* to the parent node of *n*. The equations for the two lines are:

$$L_1: \; y = C_1 \, x \tag{4.1}$$

$$L_2: \; y = \left[\frac{C_k - C_2}{k-2}\right] x + C_2 - 2\left[\frac{C_k - C_2}{k-2}\right] \tag{4.2}$$

By (4.1) and (4.2), we have the crossover point (CO):

$$\mathrm{CO} = \left(\frac{kC_2 - 2C_k}{(k-2)C_1 + C_2 - C_k} \right) \tag{4.3}$$

Therefore, if the number of tokens arriving at a node of a Gator network is greater than CO, then the set query approach will be used. Otherwise, the tuple query approach will be used.

4.2 Estimating the Query Costs

To obtain the crossover point of a pair <*n*, parent(*n*)> for a node *n*, we need to find three costs (C_1, C_2, and C_k: defined in Section 4.1) for the pair. Each of the three costs consists of the preparation cost and the execution cost. Generally, database products provide a query optimizer that does cost-based query optimization and shows the selected execution plan and the cost of the plan to the user. By using the output of the query optimizer we can remove the necessity of developing a module to estimate the query cost. However, the costs provided by the optimizer are the execution costs of the queries. Therefore, we need to find the preparation (verification, compilation, optimization, and cursor opening) costs of the queries through the observation during the query preparation.

Let us assume that we are calculating the crossover point for a node *n*. The procedure of determining C_1 of *n* will be as follows:

(1) Make a dummy tuple *t* of the memory node for the node *n*.

(2) Form an SQL statement by substituting *t* into TQT (*n*, parent (*n*)).

(3) Run the query optimizer with the plan explain flag on.

(4) Obtain the cost of the plan by parsing the output of the optimizer.

In step (1) above, the values of the joining attributes of the tuple need to be determined with caution when the joining nodes maintain the distribution of column values. The costs of executing steps (2) and (3) also need to be measured. In fact, the

cost of step (3) needs to be separated into the query preparation cost and the cursor open cost if dynamic SQL is to be used in the tuple query approach. Parsing the plan explanation output by the optimizer is not only time consuming but also not a streamlined implementation style. Therefore, we recommend that future database product implementers provide an API interface function for obtaining the cost of the chosen query execution plan. C_1 will be the summation of the costs of steps (2) and (3) and the plan cost found in step (4). However, the calculation method will vary depending on the situation.

The procedure of determining C_2 of n will be as follows:

(1) Make a dummy relation TR with two dummy tuples of the memory node for the node n.

(2) Form an SQL statement by substituting TR into SQT (n, parent (n)).

(3) Run the query optimizer with the plan explain flag on.

(4) Obtain the cost of the plan by parsing the output of the optimizer.

C_2 will be the summation of the costs of steps (2) and (3) and the plan cost found in step (4). The procedure of determining C_k of n will be the same as the procedure for determining C_2, except the number of tuples in TR is k instead of 2.

4.3 Determining Query Approach Type

In Section 4.2 the implicit assumptions are that the tokens are in main memory for the tuple query approach and that the tokens are in a database table for the set query approach. However, in reality, two other scenarios are possible: a token set larger than the crossover point is stored in main memory, not in a database table, and a token set

smaller than the crossover point is stored in a database table, not in main memory. This is true, because we estimate the appropriate query approach type for a node, *p*, and store the tokens that will be propagated into *p* accordingly (see Figure 4.3 for more details). This is why we need to consider the cost of the location change of the token set before we can determine the query approach type.

We are going to associate three functions to each memory node. The first function calculates the *cost difference between the tuple query approach and the set query approach*. When the token set is larger than the crossover point, this function returns the cost advantage of the set query approach over the tuple query approach. When the token set size is smaller than the crossover point, this function returns the cost advantage of the tuple query approach over the set query approach. We will call this function as *cost_difference*. *Cost_difference* is found by subtracting the cost of the set query (line L_2 of Figure 4.1) from the cost of the tuple query (line L_1 Figure 4.1) and taking the absolute value of it. By (4.1), (4.2), and (4.3), *cost_difference* is the value of formula (4.4) where *x* is the token set size.

$$\left| \left(C_1 - \frac{C_k - C_2}{k-2} \right) x - \left(\frac{kC_2 - 2C_k}{(k-2)C_1 + C_2 - C_k} \right) \left(C_1 - \frac{C_k - C_2}{k-2} \right) \right| \qquad (4.4)$$

The second function returns the *cost of converting a set of tokens in main memory into a database table*. We will call this function as *mm_to_table_cost*. This function can be calculated using a similar method to the one used in calculating the crossover point. That is:

- Estimate the cost of creating a temporary table and inserting a tuple into it. Let this cost be S_1.

- Estimate the cost of creating a temporary table and inserting *k* tuples into it. Let this cost be S_k.
- Formulate a linear function based on S_1 and S_k found above.

The linear function receives a number of tokens as a parameter and returns the cost. Furthermore, a common function can be shared among all of the nodes with tuples of the same length.

The third and last function returns the *cost of loading the tokens in a database table into the main memory*. We will call this function as *table_to_mm_cost*. This function can be calculated using a similar method to the one used in calculating *mm_to_table_cost*. That is:

- Estimate the cost of fetching a tuple from a temporary database table. Let this cost be F_1.
- Estimate the cost of fetching *k* tuples from a temporary database table. Let this cost be F_k.
- Formulate a linear function based on F_1 and F_k found above.

The linear function receives a number of tuples as a parameter and returns the cost. A common function can also be shared among all of the nodes with tuples of the same length.

We are assuming that all the tokens arriving at an α node are in main memory and not in the database table. Hence, when a token set, *S*, arrives at an α memory node, *n*, the query approach type for *n* will be determined by the algorithm (Figure 4.2) written in SPARKS [41].

if (size(*S*) > *n*.CO) **and**

((*n.cost_difference*(size(*S*)) > *n.mm_to_table_cost*(size(*S*)))

then create a temporary table out of the tokens in the main memory token set

use set query approach

else use tuple query approach **endif**

Figure 4.2: The decision of a query approach type

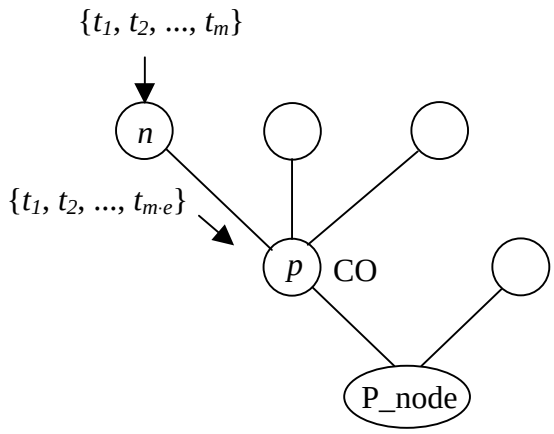


Figure 4.3: Initial query approach type for a β node

Let a β node, *p*, be the parent node of *n*. To determine the location to store the result of a query/queries invoked to propagate tokens to *p* from *n*, we need to estimate the appropriate query approach type for *p*. See Figure 4.3 where a set, *S*, of *m* tokens arrives at *n*. Let *e* be the estimated number of tokens that will be inserted into *p* per token arriving at *n*. If the total number of token insertions into *p* due to S, *m·e*, is greater than

CO of *p*, then storing the result of the join(s) of the tokens in S into a temporary table will be more efficient. This is because we can save the time to access the result tuples using a cursor and insert them into a temporary table to prepare the set query approach. Otherwise (when $m \cdot e$ is less than or equal to CO of *p*), we will accumulate the result into main memory to prepare the tuple query approach.

Table 4.1: Determining the query type for a β node

case	initial (estimated)		size of actual result	final query type
	condition	query type		
1	*in_token* × *est_token* ≤ CO	tuple	≤ CO	tuple
2			> CO	**if** (*cost_difference* > *mm_to_table_cost*) **then** set **else** tuple
3	*in_token* × *est_token* > CO	set	< CO	**if** (*cost_difference* > *table_to_mm_cost*) **then** tuple **else** set
4			≥ CO	set

Table 4.1 shows how final query type for a β node is chosen depending on the actual number of tokens propagated into the node and the values of three cost functions of the node:

(1) Cost difference between the tuple query approach and the set query approach (*cost_difference*).

(2) Cost of converting a set of tokens in main memory into a database table (*mm_to_table_cost*).

(3) cost of loading the tokens in a database table into the main memory (*table_to_mm_cost*)

In Table 4.1, *in_token* represents the number of tokens inserted into a child node *c* of a β node n, and *est_token* represents the estimated number of tokens that will be propagated into the β node per incoming token into *c*. Depending on the initial (estimated) query type, the tokens that are going to be inserted into the β node will be stored in main memory or in a temporary table. The column *size of actual result* specifies the comparison between the actual number of tokens that are being propagated into the β node and the CO of the β node. The final column *final query type* shows the query type determined after considering the benefit and the extra cost.

4.4 Token Set Propagation Algorithms

In this section, we present the algorithms that can be used in propagating token sets that arrive at a memory node, *n*, to the parent node of *n*. The procedure *start_propagation* (Figure 4.4) is invoked against a β node. Among the tokens in a CTS (see Section 7.4 for more details about the CTS), the + tokens that arrive at the same α node, *n*, can be stored into a table and be propagated to the parent node of *n*, using the *set query approach*. In that case, *start_propagation* is invoked against an α node. Therefore, the first parameter, *n*, of *start_propagation* is either an α node or a β node.

```
procedure start_propagation (n, S)
  // n: an α or β memory node //
  // S: a + token set arriving at n, an in-memory structure //
  // Action: conditionally converts S into a stored table and calls propagate //
  (2)  if n is a P-node then
  (3)        execute the rule action associated with n using the tokens in S
  (4)  else if (size(S) ≥ n.CO) and // size(S): the number of tuples in S
  (5)              (n.cost_difference (size(S)) ≥ n.mm_to_table_cost (size(S)))
  (6)              then convert S into a database table
  (7)        endif
  (8)        propagate (n, S)
  (9)  endif
end start_propagation
```

Figure 4.4: Start the propagation of a token set

The *start_propagation* accepts two parameters: a memory node, *n*, and a + token set, *S*, that arrives at *n*. If the size of *S* is larger than the crossover point of *n* and the benefit (*cost_difference*) is estimated to exceed the cost (*mm_to_table_cost*), then *S* is transformed into a database table. Then, the procedure calls the procedure *propagate* which is a recursive procedure and calls two procedures: *prepare_set* and *prepare_tuple*.

```
procedure propagate (n, S)
  // n: a memory node; let p be the parent node of n //
  // S: a + token set stored either in main memory or in a database table //
  // S2: a + token set, stores the tokens that will be propagated to p //
  // Action: inserts S into n and propagates S to p //
  (1)  insert S into n // use INSERT ... VALUES or INSERT ... SELECT //
  (2)  if (size(S) * n.est_token) > p.CO then
  (3)        prepare_set (n, S, S2)       // case 3 and 4 (of Table 4.1) //
  (4)  else  prepare_tuple (n, S, S2)     // case 1 and 2 //
  (5)  endif
  (6)  if p is a P-node then
  (7)      execute the action of the rule for n using the tokens in S2
  (8)  else
  (9)      if  (S2 in m.m.) and (size(S2) > p.CO) and // case 2 //
(10)           (p.cost_difference (size(S2)) > p.mm_to_table_cost (size(S2))) then
(11)           convert S2 into a database table
(12)      endif
(13)      if  (S2 is a stored table) and (size(S2)  < p.CO) and // case 3 //
(14)           (p.cost_difference (size(S2)) > p.table_to_mm_cost (size(S2))) then
(15)           convert S2 into a m.m. structure
(16)      endif
(17)      propagate (p, S2)
(18)  endif
end propagate
```

Figure 4.5: Insert and propagate a token set

```
procedure prepare_set (n, S, S2)
  // n: a memory node; let p be the parent node of n //
  // S: a + token set that arrives at n //
  // S2: a + token set, a database table, result parameter //
  // Action: finds tokens that will be propagated to p and stores them into S2 //
  (1)  initialize a database table S2      // Prepare set query. //
  (2)  if S is in m.m. then
  (3)       for each token t in S do
                 // Let Q be an SQL statement that is obtained by substituting t //
                 // into TQT (n, p). //
  (4)            run "Insert into S2 Q"  // execute tuple query, prepare set query //
  (5)       repeat
  (6)  else  // S is a database table. //
            // Let Q be an SQL statement that is obtained by substituting TS //
            // into SQT (n, p). //
  (7)       run "Insert into S2 Q"  // execute set query, prepare set query //
  (8)  endif
end prepare_set
```

Figure 4.6: Preparing a set query approach

The procedure *propagate* (Figure 4.5) accepts two parameters: a memory node *n*, a + token set, *S*, that arrives at *n*. It inserts the tokens in *S* into *n*. If the estimated number of tokens that will be propagated to the parent node of *n*, *p*, is greater than the crossover point of *p*, then prepare a set query approach. Otherwise, prepare a tuple query approach. Later, when the token set that will be propagated to *p* is found, the procedure re-evaluates the efficiency of the original query approach type and determines whether to hold on the approach type. If it decides not to hold on the original query type, then it changes the location of S_2. Finally, it calls itself recursively passing the appropriate parameters.

The procedure *prepare_set* (Figure 4.6) accepts three parameters: a memory node, *n*, a + token set, *S*, that arrives at *n*, and a result parameter, S_2. The parameter S_2 is a database table. The procedure finds the tokens that will be propagated from *n* to *p*, stores them into S_2, and return S_2 to the calling procedure.

The procedure *prepare_tuple* (Figure 4.7) accepts three parameters: a memory node, *n*, a + token set, *S*, that arrives at *n*, and a result parameter, S_2. The parameter S_2 is an in-memory structure. The procedure finds the tokens that will be propagated from *n* to *p*, stores them into S_2, and returns S_2 to the calling procedure.

4.5 Summary and Further Studies

To propagate multiple tokens that are arriving at a β node simultaneously, either the *tuple query approach* or the *set query approach* can be used. The *set query approach* is beneficial for a large token set while the *tuple query approach* is good for a small token set.

```
procedure prepare_tuple (n, S, S2)
 // n: a memory node; let p be the parent node of n //
 // S: a + token set that arrives an n //
 // S2: a + token set, an in-memory structure, result parameter. //
 // Action: finds tokens that will be propagated to p and stores them into S2 //
 (1)  initialize S2 in main memory  // prepare tuple query //
 (2)  if S is in m.m. then
 (3)       for each token t in S do
                 // Let Q be an SQL statement that is obtained by substituting t //
                 // into TQT (n, p). //
 (4)             result ← run Q  // execute tuple query //
 (5)             append result to S2 // prepare tuple query //
 (6)       repeat
 (7)  else  // S is a database table. //
           // Let Q be an SQL statement that is obtained by substituting S //
           // into SQT (n, p). //
 (8)       result ← run S2  // execute set query //
 (9)       put result into S2 // prepare tuple query //
(10)  endif
end prepare_set
```

Figure 4.7: Preparing a tuple query approach

In this chapter, we propose a method that determines the crossover point between the large sets and the small sets. Our method uses interpolation on the costs that can be obtained by parsing the query plans produced by the query optimizer. In relation to this, we are recommending that future database product implementers provide an API interface function to obtain the cost of the chosen query execution plan. We also propose a strategy that dynamically decides between the tuple query approach and the set query approach in propagating tokens that arrive at a β node. The proposed strategy uses three functions: *cost_difference*, *mm_to_table_cost*, *table_to_mm_cost*.

To help the implementation of the strategy, we present a set of algorithms that can be used in propagating token sets that arrive at a memory node. The algorithms consist of four procedures: *start_propagation*, *propagate*, *prepare_set*, and *prepare_tuple*. The strategy can be applied starting from either an α node or a β node.

Adelberg et al. [4] used the *Forced Delay recomputation algorithm* in maintaining derived data to reduce the cost. They exploited *update locality* to improve transaction response time. Similarly, if multiple tokens apply to the same tuple were merged into a single token, then we believe that the system performance will be further increased.

CHAPTER 5
PARALLEL EXECUTION OF SQL STATEMENTS

Today, many systems operate in an environment where multiple processors work in parallel to provide services. Among the parallel architectures, symmetric multiprocessing computers (SMPs) are widely [54] used and are closely related to the traditional single-CPU processors. Accordingly, many database products provide features to exploit the power of SMPs in executing SQL statements. The resources that allow the parallel query execution and make higher throughput possible are called parallel resources. The parallel resources include CPU, memory, and disk I/O.

While TriggerMan is in execution, it generates many SQL statements automatically (see Chapter 4). Therefore, the tuning of parallel execution of the statements is very important for the performance of the system. The parallel execution of SQL statements includes the parallel scanning of tables and the parallel processing of multiple smaller processing units. The smaller processing units are parts of the original query and can be computed independently. They can be parts of the operation of join, sort, aggregation, etc. The performance of an individual query can increase when the tables accessed from it are partitioned across multiple disks.

For example, let us assume α_2 in Figure 5.1 is partitioned on column y ($\neq x$) and no index is defined on column x. When a token t arrives at α_1, we need to do a linear

search of all partitions of α_2 to find the tuples that match with *t*. In this case, we can increase the scanning speed by reading the partitions of α_2 in parallel.

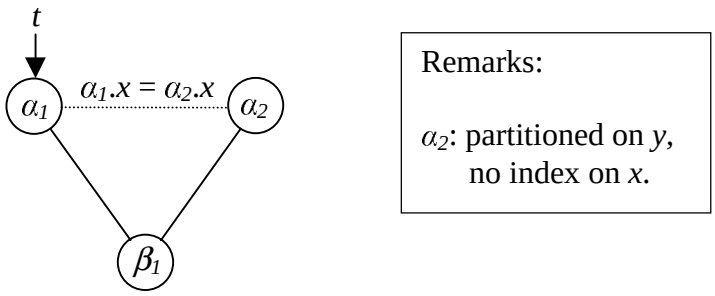


Figure 5.1: Example of a parallel scanning benefit

The parallel execution of SQL statements naturally exploits data-level concurrency and will lead to increased scalability of TriggerMan (the scalability issue is mentioned further in Chapter 6). The number of processors employed for a computation is known as the degree of parallelism (DOP). To fully utilize the parallel resources, the appropriate parallel execution strategy (the DOP and whether parallel scanning will be employed or not) for each SQL statement needs to be selected. Inappropriate strategy selection can lead to poor system performance [46].

In this chapter, we will introduce a method that finds the best parallel execution strategy for an SQL statement. To develop a generally applicable method we studied the parallel features of three database products. They are explained in Section 5.1. The parameters that are used in calculating the DOP of an SQL statement are listed in Section 5.2. The parallel execution strategy for an SQL statement is introduced in Section 5.3. Finally, a summary is given in Section 5.4.

5.1 Parallel Features of Three Database Products

Instead of developing a parallel feature utilization method for a specific database product, we are trying to develop a method that is applicable to any database product. To do that, we had to find the common parallel features of various database products first. This section compares the parallel features of three database products. Table 5.1 shows a comparison of the parallel features of three widely used object-relational or relational database products: Informix Dynamic Server with Universal Data Option (UDO) [46] (we will call this Informix/UDO), Oracle8 Enterprise Edition (EE) [64], and DB2 universal database V.5 [44].

In Informix/UDO, PDQpriority specifies the percentage of parallel resources for a query, an application, or an instance. In Oracle8, users can determine how aggressively the optimizer will attempt to parallelize a given execution plan of a query. IBM DB2 allows users to choose the DOP for an SQL statement. All three products support parallel operation at the intra-query level. The parameter that is used to enable the parallel features of a specific database product determines the number of processors that will be employed for an SQL statement. Conversely, if we know the appropriate DOP for an SQL statement, we can get the value of the parameter used by any database product. For example, a DOP can be translated into what is known as PDQpriority of Informix/UDO.

5.2 Parameters Needed to Calculate the DOP of an SQL Statement

The parameters relevant to the calculation of the DOP of an SQL statement when TriggerMan is implemented in Informix/UDO are:

Table 5.1: Parallel features of the three database products

FACTORS	Informix / UDO	Oracle8 / EE	DB2 Universal DB v.5
Partition unit	- table, index	- table, index	- table, index, and more
Operations done in parallel	- scan - join - aggregation - sort - group - etc.	- scan - join - summarizing - sort - group - etc.	- scan - join - aggregation - set operation - etc.
Parallel feature enabling methods	- PDQpriority: degree of parallelism (DOP) and parallel scan for a query, application, or instance	- parallel hint: DOP for a table in a query (1^{st} priority) - table's defined DOP (2^{nd}) - DOP of query is max of DOPs of tables	- Degree option of pre-compile or bind: DOP for static query - special degree register: DOP for dynamic query
Distinctive Features	- scan thread reservation - actual DOP = desired DOP	- actual DOP $\leq$ desired DOP	- actual DOP $\leq$ desired DOP

- *pg_byte*: Number of bytes in a page
- *t_byte*: Number of bytes in a tuple
- *n_ tuple*: Number of tuples in a node -- across all partitions
- *n_part*: Number of partitions in a node
- *n_page*: Number of pages in a node
- *r_row*: Expected number of tuples in the result
- *r_page*: Expected number of pages in the result
- *IO_per_scan_proc*: Number of pages of a Gator network node that need to be read by a scan process
- *IO_spd*: Effective number of tuples read by all the scan processes during one page I/O time, $\left\lceil \frac{r\_row}{IO\_per\_scan\_proc} \right\rceil$
- *CPU_spd*: Number of tuples that can be processed (tested against a selection condition and/or a join condition) during one page I/O time. *CPU_spd* is ∞ when no condition exists
- *spd_ratio*: Ratio of the tuple reading speed to the tuple processing speed, $\left\lceil \frac{IO\_ spd}{CPU\_spd} \right\rceil$
- *fanout*: Fanout of a node in a B^+-tree

5.3 Parallel Execution Strategy for an SQL Statement

The parallel execution strategy for an SQL statement consists of two parts: whether parallel scan will be employed or not and the DOP. The employment of parallel scan for an SQL statement depends on the necessity of parallel scan for the Gator network nodes that are accessed from the statement. If at least one node that is accessed by an SQL statement is to be scanned in parallel, then the statement will be scanned in parallel. Otherwise, it will be scanned in serial. This is because an SQL statement can access multiple nodes; nevertheless, the SQL statement is the finest level at which we can tune the parallel features of a DBMS product (see Table 5.1). In Informix/UDO, the actual number of scan processes reserved for the SQL statement will be determined by the number of partitions in the nodes that are accessed from the statement.

When a Gator network node is partitioned, if it is certain that the target tuples are spread across multiple partitions, parallel scan will be employed. The conditions where it is known that the target tuples will be spread across the partitions include:

- The data partitioning is done in a round robin fashion.
- The node is the base table of an α-node to be primed; if the selection condition is defined on the node, then it should be on a different column (or set of columns) from the partitioning column.
- The node is the temporary table created to process a set of tokens arriving at a node using the *set query approach* (explained in Chapter 4).

The DOP for an SQL statement depends on both the tuple I/O speed (*IO_spd*) of the nodes that are accessed from the statement and the costs of the functions (*CPU_spd*) that are embedded in the selection or join condition of the statement. In short, we will

employ multiple CPUs for an SQL statement when expensive (*CPU_spd* is slower than *IO_spd*) functions are embedded in the conditions of the statement. All factors that determine the DOP are properties of a Gator network node; in fact, the properties of a node interact and determine the DOP of the node. The properties of a node are:

- *IO_spd* -- determined by the node properties like tuple size, the existence of an index, the number of partitions, etc.
- Cost of functions in the selection condition (a part of *CPU_spd*) -- this is a property of a node because each selection condition belongs to a node.
- Cost of functions in the join condition (a part of *CPU_spd*) -- this can be thought of as a property of a node, because join of three or more tables is performed by joining two tables at a time in most database systems [63].

As was mentioned earlier, an SQL statement can access multiple nodes, and the finest level at which the parallel feature of a DBMS product can be tuned is the SQL statement (see Table 5.1). Therefore, we need to determine the DOP of the SQL statement out of the DOPs of the nodes accessed from the statement. We will use the largest of the DOPs of the nodes as the DOP of the SQL statement. By doing this we can maximize the processing speed of the SQL statement. This method is analogous to the one used in Oracle8 [64] (refer to Table 5.1) where the DOP of a query is the largest of the DOPs of the tables that are accessed from the query.

Now, the problem of determining the DOP of an SQL statement is reduced to determining the DOP of a node. We will determine the DOP of a Gator network node based mainly on the *spd_ratio* of the node. The *IO_spd* of a node partially determines the *spd_ratio* of the node (see Section 5.2). *The IO_per_scan_proc* partially determines

IO_spd (see Section 5.2). *IO_per_scan_proc* depends on the type of index on the access column of the node (clustered index, non-clustered index, or no index). *IO_per_scan_proc* also depends on whether the node is partitioned or not and whether parallel scan is employed or not (partitioned & parallel scan, partitioned & serial scan, and non-partitioned where serial scan is the only choice). Therefore, we need to consider the combinations of the values of those factors of a Gator network node to calculate *IO_per_scan_proc* of the node. The following paragraphs will explain eight different formulas of *IO_per_scan_proc* for the whole combination.

Formula 1 -- when clustered index is defined, node is partitioned, and parallel scan is employed

In this case, we assume that the needed tuples are evenly distributed across the partitions. Hence, the number of data pages that need to be read by a scan process is:

$$\left\lceil \frac{r\_row \times t\_byte}{pg\_byte \times n\_part} \right\rceil \tag{5.1}$$

The number of index pages that need to be read by a scan process is:

$$\left\lceil \log_{fanout} \frac{n\_tuple}{n\_part} \right\rceil \tag{5.2}$$

From (5.1) and (5.2), *IO_per_scan_proc* is:

$$\left\lceil \frac{r\_row \times t\_byte}{pg\_byte \times n\_part} \right\rceil + \left\lceil \log_{fanout} \frac{n\_tuple}{n\_part} \right\rceil$$

Formula 2 -- when clustered index is defined, node is partitioned, and serial scan is used

In this case, the number of index pages that need to be read by a scan process is the same as (5.2) and the number of data pages that need to be read is:

$$\left\lceil \frac{r\_row \times t\_byte}{pg\_byte} \right\rceil \tag{5.3}$$

From (5.2) and (5.3), *IO_per_scan_proc* is:

$$\left\lceil \frac{r\_row \times t\_byte}{pg\_byte} \right\rceil + \left\lceil \log_{fanout} \frac{n\_tuple}{n\_part} \right\rceil$$

Formula 3 -- clustered index is defined, node is non-partitioned

In this case, the number of data pages that need to be read by a scan process is the same as (5.1) and the number of index pages that need to be read is:

$$\lceil \log_{fanout} n\_tuple \rceil \tag{5.4}$$

From (5.1) and (5.4), *IO_per_scan_proc* is:

$$\left\lceil \frac{r\_row \times t\_byte}{pg\_byte \times n\_part} \right\rceil + \lceil \log_{fanout} n\_tuple \rceil$$

When a non-clustered index is defined on the access column, not all of the tuples in the pages read will be tested against the conditions that exist. To obtain the number of pages that need to be read, we are using the *Yao* approximation [75]. Given n tuples grouped into m pages, if k tuples are randomly selected from the n tuples, the expected number of pages touched is:

$$Yao\ (n, m, k) = m \times \left(1 - \prod_{i=1}^{k} \frac{nd - i + 1}{n - i + 1} \right) \text{where } d = 1 - 1/m$$

Formula 4 -- when non-clustered index is defined, node is partitioned, and parallel scan is employed

In this case, the *IO_per_scan_proc* is:

$$Yao\left(\frac{n\_tuple}{n\_part}, \frac{n\_page}{n\_part}, \frac{r\_row}{n\_part}\right) \times (1+1)$$

In the above formula, (1+1) is for the leaf level index page and the data page (the I/O for the index pages other than the leaf node is ignored).

Formula 5 -- when non-clustered index is defined, node is partitioned, and serial scan

In this case, the *IO_per_scan_proc* is:

$$Yao\left(\frac{n\_tuple}{n\_part}, \frac{n\_page}{n\_part}, r\_row\right) \times (1+1)$$

Formula 6 -- when non-clustered index is defined, and node is non-partitioned

In this case, the *IO_per_scan_proc* is:

$$Yao(n\_tuple, n\_page, r\_row) \times (1+1)$$

When no index is defined, the node needs to be scanned sequentially.

Formula 7 -- when no index is defined, and node is partitioned

Since no index is defined, *IO_per_scan_proc* remains the same whether parallel scan is employed or not. It is:

$$\frac{n\_page}{n\_part}$$

Formula 8 -- when no index is defined, and node is non-partitioned

In this case, the *IO_per_scan_proc* is:

$$n\_page$$

Now we have *IO_per_scan_proc* of a Gator network node for each value combination of factors. Hence, the *IO_spd* of a Gator network node can be obtained (see Section 5.2). We assume that the cost of each function in selection and join conditions of a Gator network node can be obtained by referring to the system catalog. We further assume that the function costs are expressed in the unit of a page I/O time or they can be transformed into the page I/O time unit. Using those function costs, *CPU_spd* of a node can be obtained. Since we have the *IO_spd* and the *CPU_spd* of a node, we can get the *spd_ratio* of the node.

To employ *n* CPUs effectively, at least *n* units (pages or tuples) need to be processed, so that each CPU can handle at least one unit. Hence, when a clustered index is defined on the access column of a Gator network node, the DOP of a Gator network node is:

$$\min (r\_page, spd\_ratio)$$

When a non-clustered index is defined on the access column, the DOP is:

$$\min (r\_row, spd\_ratio)$$

Otherwise (when no index is defined), the DOP is:

$$\min\left(\frac{n\_page}{n\_part}, spd\_ratio\right)$$

Now, we have obtained the DOP of a Gator network node. The DOP of an SQL statement is the largest of the DOPs of the nodes accessed from the statement.

5.4 Summary

In this chapter, we introduced a method of tuning the parallel features of the host DBMS for the execution of the SQL statements initiated from the TriggerMan system. To develop a parallel feature tuning method generally applicable to any database product, the parallel features of three database products have been studied. The parallel execution strategy (the appropriate DOP for parallel processing and when to use a parallel scan) for an SQL statement was developed based on the parallel execution strategies for the nodes that are accessed from the statement. The properties of a node that were considered in determining the parallel execution strategy for the node are:

- Partitioning of the node and the scheme of partitioning.
- Presence of any index on the access column and the clustering of the index.
- Costs of the functions in the selection and/or join conditions that are associated with the node.

The conditions that make parallel scan beneficial were introduced. We also provided the formulas that calculate the appropriate DOP for a Gator network node. The execution strategy for an SQL statement was determined using the following policies:

- An SQL statement is processed in parallel if at least one node that is accessed from the statement needs to be scanned in parallel.
- The largest of the DOPs of the nodes that are accessed from an SQL statement is used as the DOP of the statement.

CHAPTER 6
SCALABILITY

Scalability can be defined by how well a solution to a problem will work when the size of the problem increases. The TriggerMan system will have a good scalability, if it performs well utilizing a given hardware environment against the increase of the number of triggers, incoming tokens, etc. There are many reasons why the TriggerMan system has to be scalable [38]. Among them, some major reasons are:

- The convenience of writing applications - Trigger systems can be used for active information processing for large sets of triggers instead of writing custom applications to carry out triggering logic.
- The efficiency and the convenience of active information distribution - the critical information to the survival of an enterprise can be delivered to the client as soon as it is available without excessive waiting of the client.
- The enormous growth rate of Web-based applications [48] - users can create a large number of triggers via the interfaces provided by the applications.
- The development of parallel computers, especially SMP machines - to fully utilize the available parallel machine, the TriggerMan system needs to be scalable.

A study on parallel implementation of rule-based systems shows that only a limited speed-up can be obtained [31],[49]. The reasons of small speed-up in [31] are: (1)

the small number of rules relevant to each change to data memory; (2) the large variation on the processing requirements of relevant rules; and (3) the small number of changes made to data memory between synchronization steps. However, in TriggerMan the reasons do not necessarily hold: (1) the number of triggers relevant per change to data source is dependent of the total number of triggers; (2) the variation on the processing requirements of relevant triggers can be limited intentionally; and (3) in principle, there is no synchronization step in TriggerMan. This gives a hope that TriggerMan can have higher parallel speed-ups than observed in parallel production system.

Other properties of TriggerMan that increase the possibility of high speed-up through parallel processing are:

- Multiple triggers can have exactly the same conditions but different actions.
- The size of a data source (database) is virtually unlimited.
- Change rates of data sources can be extremely high; changes are made by external applications.
- Users want to define as many triggers as possible.

Therefore, the parallel processing is expected to increase the scalability of TriggerMan. To do parallel processing, we need to identify the concurrencies that can be exploited. Four kinds of concurrency that TriggerMan can exploit are:

- Token-level concurrency -- Multiple tokens can be processed concurrently (Chapter 7).
- Condition-level concurrency -- Multiple conditions can be tested against a single token concurrently (Section 6.1).

- Rule action concurrency -- Rule actions of multiple rules or multiple instantiations of the action of a single rule can be executed concurrently (Section 6.2).
- Data-level concurrency -- A set of tuples in an α or β node of a Gator network [36] can be processed in parallel (Section 6.3).

TriggerMan will maintain a task queue [38] in shared memory [46] to store incoming tokens or internally generated tasks. Basically, the task queue is the same concept as the process queue in an operating system. The task queue performs a central role in exploiting available concurrencies.

Parallel token processing (token-level concurrency) causes problems in the semantic correctness of trigger processing. As a result, we find a way to incorporate parallel token processing into an asynchronous trigger system in a productive way. This is the motive behind the introduction of consistency levels for trigger processing. These topics are covered in Chapter 7. The following sections will explain how the other three concurrencies could be exploited.

6.1 Condition-level Concurrency

The two kinds of conditions in a trigger condition are the selection condition and the join condition. The join conditions are checked when the tokens are propagated through Gator networks. Therefore, the join conditions are checked concurrently as the tokens are processed concurrently (Chapter 7). The selection conditions are checked concurrently by the normalized selection predicate index (SPI) structure (Figure 6.1) [40],[57]. The normalized SPI is made by applying common sub-expression elimination.

See [38] and [40] for more details on the SPI. If a triggerID set is large, parallel processing of the triggerID set can be achieved by partitioning the set into multiple equal-size subsets (partitions) and processing the partitions in parallel [38]. If the constant set for an expression signature is large, then it too can be partitioned and processed in parallel.

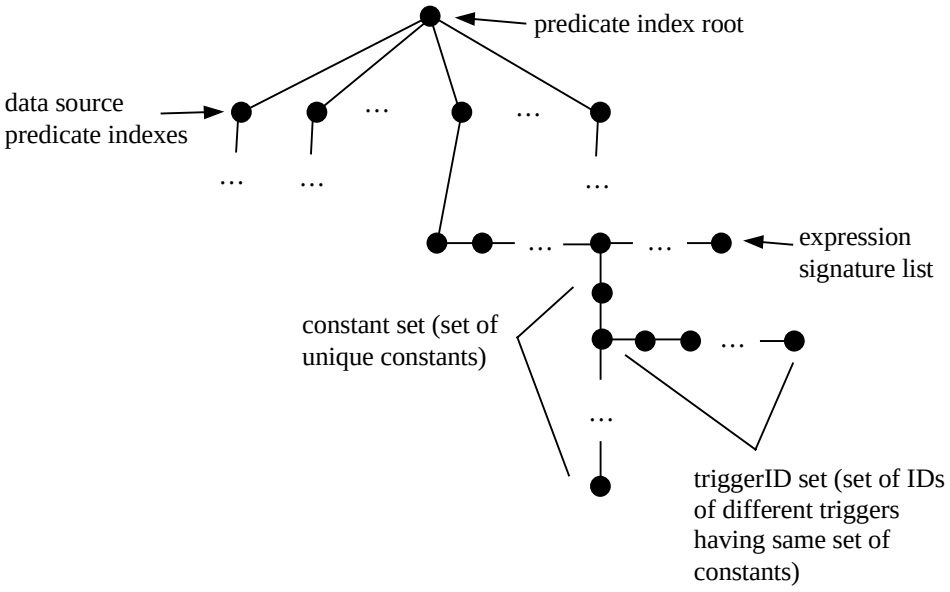


Figure 6.1: Normalized Selection Predicate Index structure

When a new token arrives, it will be inserted into the task queue, then it will be passed to the root of the predicate index (which locates its data source predicate index). All of the constant sets of the data source need to be searched in order to find the triggerID sets that match the current token. After the triggerID sets are found, each trigger in each set need to be processed for additional conditions testing and action execution.

6.2 Rule Action Concurrency

If a token satisfies the conditions of multiple triggers, then executing the actions in parallel can increase the speed of TriggerMan. Since the normalized predicate index is used in TriggerMan, the triggers that are simultaneously satisfied must be in a triggerID set. By enqueueing the actions of each trigger into the task queue, we can execute the actions in parallel. However, according to Amdahl's Law [65], the maximum speedup is limited by the speed of serial operations (operations that need to be performed by one process). The serial operations include enqueueing the tasks into the task queue, and dequeueing them from the queue, and substituting the placeholders in the trigger definition by the values from the (compound) tokens.

Therefore, we need to incorporate triggerID set partitioning into TriggerMan. This partitioning is also related to condition-level concurrency (Section 6.1). The idea of triggerID set partitioning is depicted in Figure 6.2. When a token matches the constant related with a partitioned triggerID set, the set is enqueued into the task queue partition by partition. After a triggerID partition is dequeued from the task queue, the triggers in a partition are processed sequentially using the procedural control.

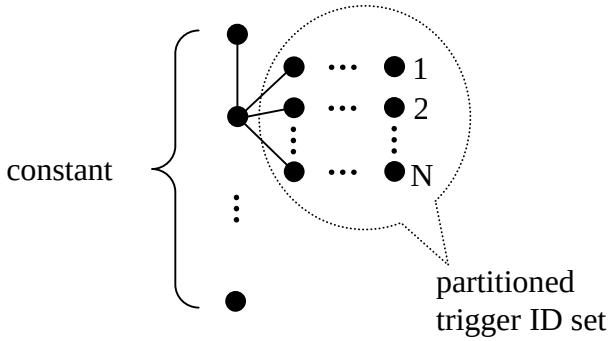


Figure 6.2: Partitioned triggerID set

Determination of when and how to partition the triggerID set are optimization problems and left for further study. The factors related with these optimization problems are:

- Overhead of enqueueing the task into the task queue -- minimize it.
- Degree of parallelism (number of partitions) -- maximize it.
- Execution path length of each partition -- keep it under a threshold.

6.3 Data-level Concurrency

Data-level concurrency can be exploited if we divide large tables into multiple partitions and utilize the parallel database query feature provided by the host DBMS and the underlying computer [21],[46]. In Chapter 5, we proposed a method of tuning the parallel features of the host DBMS for the efficient execution of SQL statements. When and how to partition a large Gator network node is left as a further study.

6.4 Summary and Further Studies

In this chapter the properties of a trigger system that shows the possibility of high speed-up through parallel processing were examined. We identified four kinds of concurrencies that exist in a trigger system and can be exploited to increase the speed-up of the system. They are condition-level concurrency, rule action concurrency, data-level concurrency, and token-level concurrency.

In TriggerMan, a normalized SPI (selection predicate index) structure is used in finding the α nodes at which a token arrives. A token and an α node pair is enqueued in the shared task queue as a task. By letting multiple processors process the tasks in the

queue in parallel, the condition-level concurrency and the rule action concurrency can be exploited. The utilization of token-level concurrency is discussed in Chapter 7.

We also presented the idea of partitioning a large triggerID set in the normalized SPI structure. Through triggerID set partitioning, we can reduce the unparallelizable portion of the work, which increases the speed-up of the system. Finding when and how to partition the triggerID set is an interesting optimization problem and is left for further study.

CHAPTER 7
TRIGGER PROCESSING CONSISTENCY LEVEL AND PARALLEL TOKEN PROCESSING

In the context of discrimination network maintenance, a ***computation unit*** is an atomic unit of processing concerning token(s) against one or more memory nodes of a discrimination network. The processing of a token, *tk*, against a memory node, *n*, includes applying *tk* to *n*, joining *tk* with the siblings of *n* if necessary, and propagating *tk* or the result of joining to the parent nodes of *n*.

In this paper, we use two kinds of computation units: the *reduced computation unit* and the *expanded computation unit*. The reduced computation unit is the processing of a token against a single memory node. The expanded computation unit consists of the processing of a token against an α memory node, α_1, the processing of the tokens propagated from α_1 against the parent node of α_1, and so on. This happens until no more tokens are propagated or tokens reach at the P-node. To exploit the power of parallel computers, parallel processing of *computation units* is essential. For the sake of simplicity, *processing a token, t, against a memory node, n,* means *processing the computation unit that includes the processing of t against n*.

Basically, we assume the tokens that arrive at different discrimination networks can be processed in parallel. This will increase the performance of the system. To further increase the system performance, we believe the parallel processing of the tokens that arrive at the same discrimination network is necessary. For the sake of simplicity,

parallel token processing means *parallel processing of the tokens that arrive at the same discrimination network*. However, uncontrolled parallel token processing creates problems in the semantic correctness of trigger processing. The list of the problems are:

- Out of order execution of the multiple instantiations of the action of a single trigger (***out-of-order rule action execution***).
- Trigger action execution using a compound tuple that is created due to an *untimely joining error* (Subsection 7.1.2). The compound tuple is called a ***phantom compound tuple***.
- Failure to execute a trigger action since the system cannot detect a transient compound tuple due to an *untimely joining error* or a transient tuple (***lost transient tuple***).
- Permanent corruption of a memory node of a discrimination network (***memory node corruption***).

We conjecture that the above list of problems is complete because of the following reasons. When the P-node of a discrimination network receives all and only correct tokens, the unique problem that can occur is the *out-of-order rule action execution*. When child nodes are correct and tokens join in a timely manner, all and only correct tokens arrive an internal node. The P-node can receive incorrect tokens due to either the *untimely joining error* or the *memory node corruption*. A memory node can be corrupted either temporarily or permanently. The *lost transient tuple* problem can corrupt them temporarily. Memory nodes are *permanently corrupted* when the tokens that arrive at the same tuple are processed out-of-order or when an *untimely joining error*

occurs. The above discussion illustrates that the list of problems appears to be comprehensive. A complete proof is left as further study.

Serial token processing against a discrimination network that does not have virtual α nodes removes all the above problems and provides an exact semantic consistency in trigger processing. By serial token processing, we mean the serial processing of tokens that are delivered from the *ideal data sources*. An *ideal data source* delivers tokens in the commit order of the transactions to which the tokens belong. Among the tokens that belong to the same transaction, an *ideal data source* delivers tokens in the order of real execution.

However, to exploit the power of parallel computers and increase the performance of the system, we need to relax the strict semantic requirements in trigger processing. In other words, if some of the above problems were allowed to happen, then the system performance could be increased through parallel token processing. The trigger users would determine the appropriate semantic requirements for his/her own triggers.

In this chapter, we will introduce the consistency levels of trigger processing and the techniques to achieve them. The techniques primarily include the support for parallel token processing. Section 7.1 gives the notational conventions and the definitions of terms that are used throughout this chapter. Section 7.2 defines four consistency levels (Level 0 through Level 3) of trigger processing. Sections 7.3, 7.4, 7.5, and 7.6 introduce the techniques for achieving the levels 3, 2, 1, and 0, respectively. Section 7.7 discusses the implementation alternatives of an asynchronous trigger system. Finally, Section 7.8 summarizes this chapter.

7.1 Notational Conventions and Definitions of Terms

This section contains the definitions of terms and the notational conventions that apply to the rest of this chapter.

7.1.1 Notational Conventions

The notations used throughout Chapter 7 are as follows:

- atk , atk_1, atk_2 : atomic tokens
- tk , tk_1, tk_2 : atomic tokens or compound tokens
- +, –, δ : specify insert, delete, and update, respectively
- tp, tp_1, tp_2 : memory node tuples
- tm_1, tm_2 : timestamps
- (tp_1, tp_2), tp(tp_1, tp_2) : a compound tuple comprised of tp_1, tp_2
- atk_1(+, tp_1), atk_2(δ, tp_2) : atomic tokens with contents
- atk_3(+, tp_3: tm_3) : an atomic token with contents and timestamps
- tk_4(+, tp_4, tp_5) : compound tokens with contents
- tk_4(+, tp_4: tm_4, tp_5: tm_5) : a compound token with contents and timestamps
- α_1, α_2, α_3 : α memory nodes
- β_1, β_2, β_3 : β memory nodes
- ts(tk_1) : the timestamp of token tk_1

7.1.2 Definitions of Terms

A ***family*** is a set of objects that have the same key and are related with the same node of a discrimination network. The objects can be a token, a tuple, or a line in a *Stability Lookaside Buffer* (SLB, see sections 7.5 and 7.6).

Two of a token or an SLB line that belong to the same family are ***comparable*** if they have different timestamp vectors and each timestamp of one is greater than or equal to the corresponding timestamp of the other. When two compound tokens are not *comparable,* they are ***incomparable***.

Let tk_1 and tk_2 be an atomic token, a compound token, or an SLB line that belong to the same family. If each timestamp of tk_1 is greater than or equal to the corresponding timestamp of tk_2, then tk_1 is ***younger than*** tk_2. When tk_1 is *younger than* tk_2, we say tk_2 is ***older than*** tk_1. An atomic token or a compound token that is *younger than* any other token in the same family is called the ***youngest*** token.

If a compound token, *tk*, is propagated from an atomic token *atk,* then *atk* is the ***initiating token*** of *tk.*

When the tokens arrive from the data sources, the system accumulates them and creates a batch. A ***cycle*** is the time during which the tokens in one batch are processed. A cycle begins after the previous cycle has finished. After a cycle that executes a batch begins, no more tokens could be inserted into that batch.

When an atomic token is processed against a discrimination network node, if the creation of a compound token is inconsistent with the real world situation, then we call it an ***untimely joining error***. The two cases of *untimely joining error* are:

- The creation of a compound tuple that never existed, 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 period of time.

The ***timing error*** is the maximum timestamp difference between the two tokens that created an *untimely joining error*.

We say a memory node ***stabilizes*** if the parallel application of a set of tokens arriving at the node produces the same final content as the content that would be produced by the serial application of the same set of tokens. The term ***converge*** is also used to refer the same situation. Similarly, ***stabilization*** and ***stability*** are used to mean the same concept as ***convergence***.

An ***atomic SLB*** contains information related to the processing of the atomic tokens that arrives at an α node. A ***compound SLB*** contains information related to the processing of the compound tokens that arrives at a β node.

7.2 Trigger Processing Consistency Levels

Before the trigger processing consistency levels are defined, we will compare the consistency levels with the degrees of consistency in the transaction processing and recovery (Subsection 7.2.1). Then the criteria to define the consistency levels will be introduced (Subsection 7.2.2) and the definitions of trigger processing consistency levels will be given (Subsection 7.2.3).

7.2.1 Transaction Consistency Degrees and Trigger Consistency Levels

In a shared environment, to protect the database from inconsistencies, locking protocols are used. Responsibilities for obtaining and releasing locks can be either assumed by the user or delegated to the system. Motivated by the fact that some database systems use automatic lock protocols and provide a limited degree of consistency, four degrees of consistency were defined [27]. The lower the degree is, the more portions of the responsibilities concerning the locking are given to the user. Since the user knows the semantics of the data, fewer locks could be used when the user controls the locking. However, this can make programming difficult.

The purpose of the trigger consistency levels is to allow slight relaxation of semantic consistency to increase the performance of the system significantly. The trigger users can save expenses by choosing appropriate consistency levels for their triggers. The higher the level is, the lower the performance is and the more inconsistency problem can exist. Due to the characteristics of the asynchronous trigger system, the tokens that arrive at the system are from the committed transactions. Therefore, tokens need to be processed in the transaction commit order to leave the system in a consistent state. The degrees of consistency for a transaction are compared with the trigger processing consistency levels in Table 7.1.

In summary, the degrees of consistency for a transaction are analogous with the levels of consistency of trigger processing. However, the trigger processing consistency levels require more complex techniques to consider the token processing order, virtual α nodes, etc. Simple read/write lock protocols are not enough for the trigger processing consistency levels.

Table 7.1: The consistency degrees and the consistency levels

	Transaction consistency degrees	Trigger consistency levels
Similarity	• The user needs to protect himself against the sources of inconsistencies that could happen in the lower consistency degrees. • The purpose is to trade the increased difficulty of the programming for gains in the performance of the system.	• The user needs to protect himself against the sources of inconsistencies that could happen in the lower consistency degrees. • The purpose is to trade the consistency in the trigger action execution with the performance of the system.
Difference	• Transactions can be committed in any order. The order is determined dynamically.	• Tokens need to be processed in the commit order of the transactions to which the tokens belong.

7.2.2 Criteria of Consistency Level Definition

Among the problems of uncontrolled parallel token processing, the problems of *phantom compound tuple* and *lost tuple* can be explained using the terms of *untimely joining error* and *timing error*. The later two terms are defined in Subsection 7.1.2.

An example of untimely joining error is given in Figure 7.1 where tk_3 deletes tp_3 from α_3 and tk_4 inserts tp_4 into α_4. The token tk_3 precedes tk_4. Assume that tp_3 and tp_4 join with each other. Following serial processing, tp_3 would be deleted by tk_3 before tp_4 could join it. When two tokens are processed in parallel, if tk_4 is processed before tk_3, then tp_4 will incorrectly join with tp_3. The compound tuple that consists of tp_3 and tp_4 will be inserted into β_1 and could be used in executing the trigger action. This is an *untimely joining error* with its *timing error* of $ts(tk_4) - ts(tk_3)$.

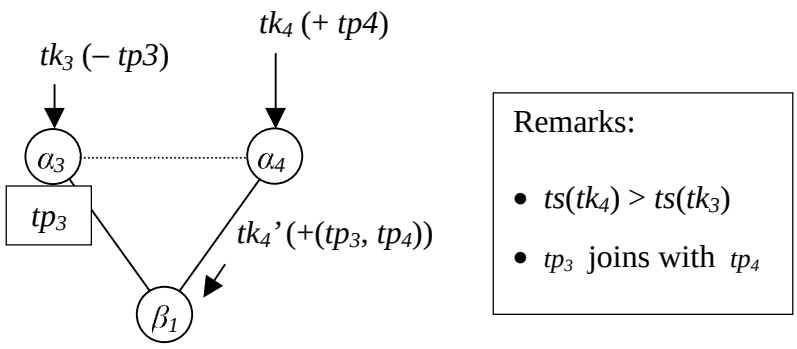


Figure 7.1: Untimely joining error

An example of *memory node corruption* is given in Figure 7.2 where tk_1 and tk_2, consecutively, arrive at α_1. When tk_1 and tk_2 are processed in parallel, if tk_2 is applied to tp_1 before tk_1, then tk_1 needs to be discarded later. Otherwise (if tk_1 is applied to tp_1 later), tp_1' will not exist after two tokens are processed, which is wrong. (Hint: Do not apply an older token after a younger token in the same family.) By discarding an older token, an α node will converge on the content that would be generated by serial token processing. The notion of convergence of a memory node plays an important role in defining the trigger processing consistency levels later in this section.

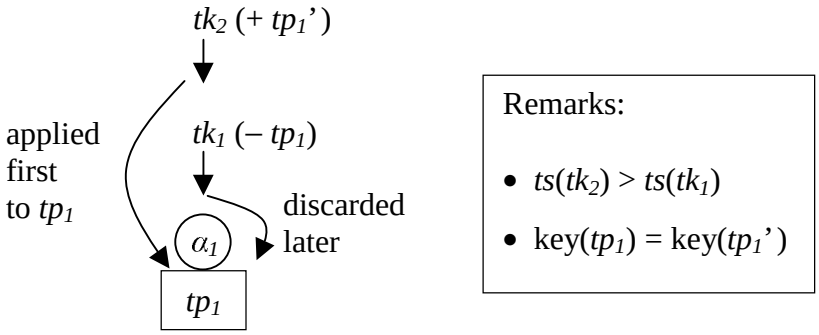


Figure 7.2: Convergence of a memory node

Among the problems of parallel token processing, memory node corruption can never be allowed. This is because the state information of the system could be totally corrupted, eventually, by allowing this problem. Three problems that have minor complications will be allowed for some consistency levels to improve the performance of the system.

From the problems of uncontrolled parallel token processing, we derived the following criteria of the consistency level definition:

- The execution order of the multiple instantiations of the action of a single trigger.
- The content or the stabilization of memory nodes.
- The existence of an *untimely joining error* in a compound tuple.
- The amount of *timing error* of an existing *untimely joining error*.

7.2.3 The Definition of Trigger Processing Consistency Levels

There are four trigger processing consistency levels: Level 0 through Level 3. The higher the level is, the less the semantic problem exists and the lower the system performance is. For example, the performance of Level 3, which is the highest level, has the lowest performance since it does not allow parallel processing of the tokens that arrive at the same discrimination network. Meanwhile, Level 0 consistency only requires the contents of memory nodes to converge and provides the highest performance.

Using the criteria given in Subsection 7.2.2, the trigger processing consistency levels may be defined as:

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

(a) The contents of the memory nodes of the discrimination network for *T* are always correct.

(b) No untimely joining error exists. That is, all compound tuples generated in the system consist of chronologically joined tuples.

(c) Multiple instantiations of the action of *T* are executed in the same order as would be done if tokens that arrive at the discrimination network for *T* were processed serially.

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

(a) The contents of the memory nodes of the discrimination network for *T* are always correct.

(b) No *untimely joining error* exists. That is, all compound tuples generated in the system consist of chronologically joined tuples.

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

(a) The contents of the memory nodes of the discrimination network for *T* converge.

(b) The *timing error* in the joining tuples is limited to some fixed value.

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

(a) The contents of the memory nodes of the discrimination network for *T* converge.

The techniques that provide levels 3, 2, 1, and 0 consistency while maximizing the performance are explained in sections 7.3, 7.4, 7.5, and 7.6, respectively.

7.3 Trigger Processing Consistency Level 3

Level 3 is the highest consistency level of trigger processing. We can provide Level 3 consistency for a trigger *T* by serially processing the tokens that arrive at the discrimination network for *T*. As a result, the expanded computation unit is implicitly used for Level 3 consistency. Level 3 consistency has the lowest performance among the four consistency levels.

By the definition of Level 3, the tokens that arrive at different discrimination networks can be processed in parallel. Assume triggers T_1 and T_2 have discrimination networks N_1 and N_2, respectively. Further assume a token tk_1 arrives at N_1 and token tk_2 arrives at N_2, consecutively. If two tokens are propagated all the way up to their P-nodes, then the action of T_2 could be executed before the action of T_1. This is because tk_1 and tk_2 can be processed in parallel by the definition of Level 3 consistency.

It is necessary to inform the users who *subscribe to* multiple triggers with Level 3 consistency about the possibility of mixed execution of the actions of multiple triggers. We say that a user *subscribes to* a trigger if he/she registers for the event that is raised by the action execution of the trigger.

When one or more virtual α nodes exist in the discrimination network for a trigger, a special technique needs to be employed to provide Level 3 consistency. This technique is based on the use of a ***shadow table*** and is explained in Subsection 7.3.1. However, the shadow table technique causes the duplicate compound token problem. To remove the duplicate compound token problem, we developed a technique that is explained in Subsection 7.3.2.

7.3.1 Support of Virtual α Nodes with Shadow Tables

The virtual α node was introduced in the A−TREAT algorithm [35] to reduce storage requirements. To join a token with a virtual α node, the base table of the virtual α node needs to be accessed. Since the base table always contains the most up-to-date tuples, the token that joins with the base table cannot see the base table content at the point of time when the token was created. Therefore, the *untimely joining error* is unavoidable when a virtual α node exists in a discrimination network.

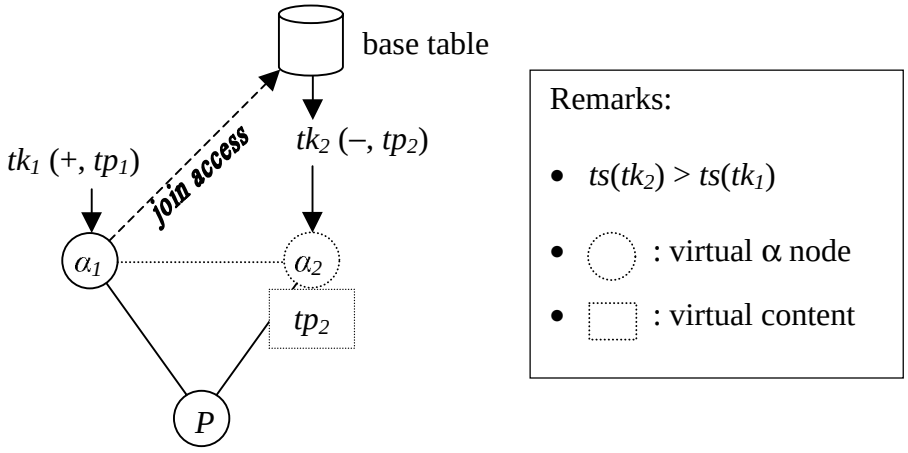


Figure 7.3: Virtual α node and an *untimely joining error*

An example is given in Figure 7.3 where a discrimination network with one stored α node, α_1, and one virtual α node, α_2, is shown. Assume tp_1 and tp_2 belonging to α_1 and α_2, respectively, join together. At this point tk_1 and tk_2 arrive consecutively at the Gator network and tk_1 is processed first. When the base table of α_2 is accessed while tk_1 is

being processed, it can occur that tp_2 of the base table has already been deleted. Hence, tp_1 cannot join with tp_2, which is an *untimely joining error*.

To remove the *untimely joining error*, we propose the idea of maintaining a copy of the base table in the state space of the system. The base table copy is called a ***shadow table***. A *shadow table* is exactly same as the base table except that it contains slightly older data. All virtual α nodes that are defined on the same base table can share the same *shadow table*. The sharing of a *shadow table* has two implications:

- The maintenance of a shadow table is cheap since only a single application of a token to the shadow table is needed irrespective of the number of virtual α nodes that share the shadow table.
- Since a shadow table could be shared among multiple discrimination networks, the modification time of the shadow table is important to all discrimination networks that share the shadow table. Occasionally, the processing of the tokens that arrive at the discrimination networks that share a common shadow table needs to be serialized. This increases the system complexity and decreases the number of tokens that can be processed in parallel. An example is given in Figure 7.4.

In Figure 7.4, tk_1, tk_2, and tk_4 arrive at two Gator networks, consecutively. Assume tk_2 satisfies the selection conditions of α_2 and α_3, tp_2 joins with tp_1, and tp_2 joins with tp_4. If tk_2 is processed before tk_1, then a compound tuple of tp_1 and tp_2 will not be created. This is an *untimely joining error*. If tk_4 is processed before tk_2, then a compound tuple of tp_2 and tp_4 will be created. This is another *untimely joining error*. To prevent both *untimely joining error*s, tk_1, tk_2, and tk_4 need to be processed consecutively.

However, if the shadow table were not shared, then tk_1 and tk_4 could be processed in parallel. The advantages and disadvantages of a shadow table can be summarized as follows:

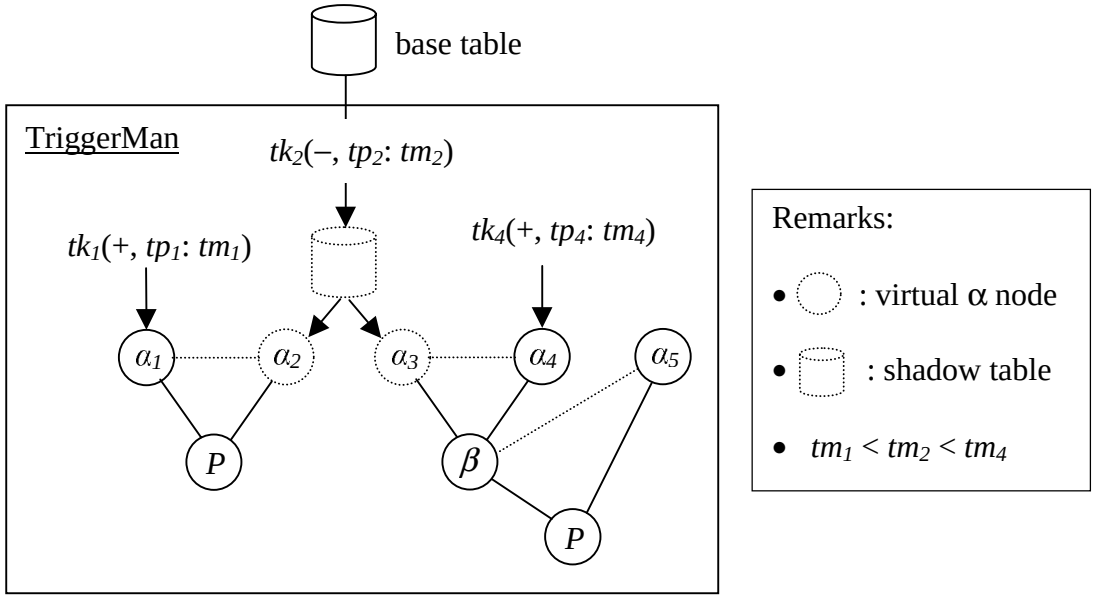


Figure 7.4: A shadow table supporting two virtual α nodes.

Advantages

1. Accessing a shadow table is faster than accessing a base table.
2. The SLB structure and algorithm to stabilize a β node is simpler with a shadow table than without a shadow table. In other words, the β node stabilization method of Strategy II (Subsection 7.5.2.2) is simpler than that of Strategy III (Subsection 7.6.4).
3. A shadow table increases the trigger processing consistency level from 0 to 1.

Disadvantages

1. It takes time to create a shadow table.
2. Storage space is needed to store a shadow table, which is usually large.
3. A shadow table needs to be maintained and it requires CPU resource.

7.3.2 Preventing Duplicate Compound Tokens

A *reflexive join* is an operation that joins a table with itself, on different columns. Assume a trigger, T_1, contains a reflexive join in its condition. If the reflexive join is implemented using virtual α nodes in the discrimination network for T, then the parent node of the virtual α nodes could receive duplicate compound tokens. The duplicate tokens would execute a trigger action more than once using exactly same data. This is called the ***duplicate compound token*** problem.

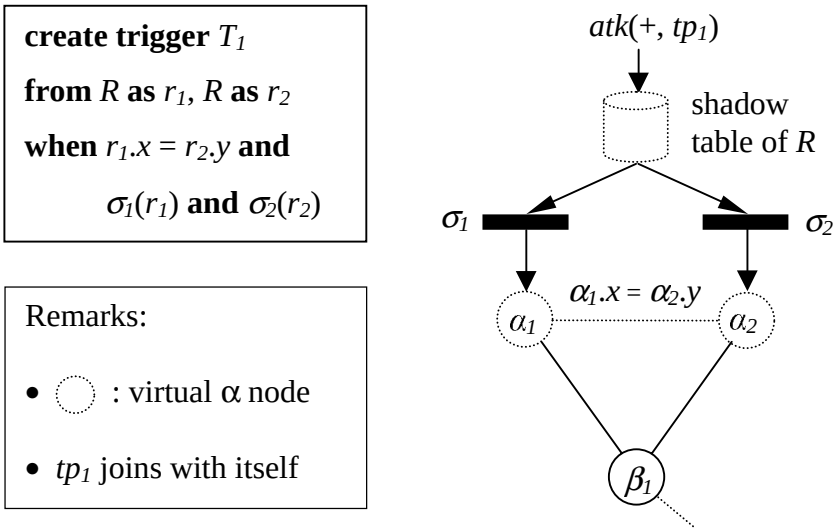


Figure 7.5: Creation of duplicate compound tokens

An example is given in Figure 7.5 where we assume tp_1 joins with itself and satisfies selection conditions, σ_1 and σ_2. The base tables of virtual α nodes are updated before the tokens are delivered to the system. Similarly, we apply the tokens from the base table *R* to the shadow table before they are propagated to α_1 and α_2. Then, when *atk* arrives at α_1, it joins with the copy of itself in the shadow table and creates a compound token that will be propagated to β_1. Later, when *atk* arrives at α_2, it joins with the copy of itself in the shadow table and creates a duplicate compound token that will also be propagated to β_1.

The creation of duplicate compound tokens has been shown. To detect and discard duplicate compound tokens, we propose the use of a ***Duplicate-Lookaside Buffer*** (DLB) that originated from the idea of an SLB (see sections 7.5 and 7.6 for the details on SLB usage). Each β node or P-node, *n*, such that *n* has two or more virtual α node children that have the same shadow table or base table, will be equipped with a DLB.

The DLB will contain a piece of information called a *line* for each tuple that joins with a copy of itself in the current cycle. Each line contains a <key, time stamp> pair. The manipulation of the DLB appears in Figure 7.6.

Let us consider an example of duplicate compound token detection. This is continued from Figure 7.5. After *atk* is processed against α_1, a compound token tk_1 (+, tp_1, tp_1) arrives at β_1. Since tk_1 succeeds the checking of Step (i) of Figure 7.6, it creates a line (<key(tp_1), $ts(tp_1)$>) in the DLB of β_1 by Step (ii). Later, when *atk* is processed against α_2, another compound token tk_2 (+, tp_1, tp_1) arrives at β_1. The token tk_2 also

successfully passes the check in Step (ii), and finds a DLB line *l* such that $ts(l) = ts(tp_1)$. Hence, tk_2 will be discarded. Now, the problem of duplicate compound token is resolved.

- A DLB is cleared after each cycle.
- The processing of the compound tokens *tk* that arrive at *n* needs to be preceded by the following two steps:
  (i) Check if *tk* is one of the duplicate compound tokens (i.e. if the initiating token *itk* of *tk* is from one of the virtual α nodes with the same shadow table or base table and appears in *tk* more than once), then go to step (ii). Otherwise, skip step (ii).
  (ii) **if** the DLB of *n* contains a line *l* with the same key as *itk* **then**
  if $ts(l) \geq ts(itk)$ **then** discard *tk*
  else change timestamp field of *l* to $ts(itk)$
  endif
  else initialize a line in the DLB using *itk*
  endif

Figure 7.6: Manipulation of DLB

7.4 Trigger Processing Consistency Level 2

Compared to Level 3, Level 2 removes the requirement of serial execution of multiple trigger action instantiations. To elaborate, when *n* tuples arrive at the P-node of the discrimination network for a trigger *T*, the *n* tuples create *n* instantiations of the action

of *T*. Level 2 consistency allows those *n* action instantiations to be processed in any order. Unless otherwise stated, each trigger that is mentioned in this section is supposed to be defined with Level 2 consistency.

To improve the performance, the tokens that arrive at the α nodes of the discrimination network for a trigger can be processed in parallel as long as they do not break the conditions of Level 2 consistency. As an example, a Gator network for trigger T_1 is shown in Figure 7.7. In the figure, *n* + tokens arrive at α_1 while no token arrives at α_2. In this case the *n* tokens can be processed in parallel. Note that the processing of the *n* tokens does not corrupt memory nodes α_1 and α_2 and does not cause any *untimely joining error*.

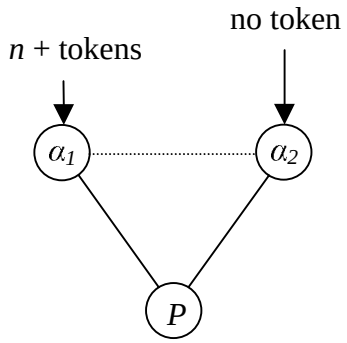


Figure 7.7: A Gator network for trigger T_1

In another case, multiple tokens that arrive at the α nodes of a discrimination network for a trigger cannot be processed in parallel. An example is given in Figure 7.8 where tokens tk_1 and tk_2 arrive at the Gator network for a trigger T_2 consecutively.

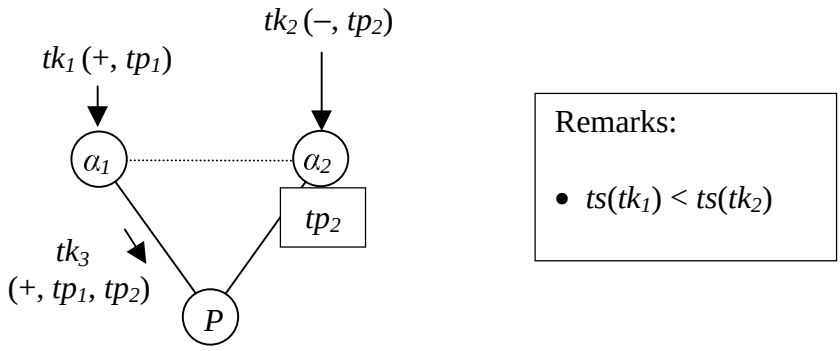


Figure 7.8: A Gator network for trigger T_2

Assume that tp_1 joins with tp_2. If tk_1 and tk_2 are processed in serial, then a compound token, tk_3(+, tp_1, tp_2), will be inserted into P. However, if tp_2 is processed before tp_1 when they are processed in parallel, then tk_3 will not be created. This is an *untimely joining error*. Therefore, to remove the *untimely joining error* and guarantee Level 2 consistency, tk_1 and tk_2 must be processed in serial.

As we discovered, among the tokens that arrive at the same discrimination network, some tokens can be processed in parallel while others cannot. Among the tokens that arrive at the same discrimination network, a set of consecutive tokens that can be processed in parallel is called a ***Concurrent Token Set*** (**CTS**). The tokens that are propagated from the tokens of a CTS can also be processed in parallel. In other words, the CTS property is inherited (***CTS inheritance***).

If the *reduced computation unit* (definition is given at the beginning of Chapter 7) is used, then the CTSs need to be recalculated at each node along the path from the α node to the P-node. To utilize the *CTS inheritance* and to avoid the overhead of CTS

recalculation, the *expanded computation unit* is used for Level 2 consistency. However, this might decrease the parallel speedup because of the increased job granularity.

Our proposed technique of detecting the CTS is presented in Subsection 7.4.1. Subsection 7.4.2 explains our proposed architecture for efficient parallel processing of tokens. Finally, Subsection 7.4.3 summarizes the techniques for processing triggers with Level 2 consistency.

7.4.1 CTS Detection

Since the *extended computation unit* is used for Level 2 consistency, only the CTSs from among the set of tokens arriving at the α nodes of a discrimination network need to be detected. The CTS detection requires finding the set of consecutive tokens that can be processed in parallel and making it as large as possible. We need to detect the CTSs for each discrimination network in the system.

An example is shown in Figure 7.9 where n tokens, tk_1, tk_2, ... , tk_n, are arriving at a discrimination network consecutively in a cycle. When $j–i+1$ tokens are in the *current CTS*, as illustrated in Figure 7.9, we test the possibility of inclusion of tk_{j+1} into the set. For tk_{j+1} to be able to be included into *current CTS*, tk_{j+1} can be processed in parallel with each token in *current CTS*.

Let N_1 be a discrimination network for a trigger, T_1. Then, the possibility of parallel processing of two tokens that arrive at N_1 depends on:

- The event types of the tokens.
- The α nodes at which the tokens arrive.

- The event clause of T_1 -- whether an event clause appears in the trigger definition and the type of the event when the event clause appears.

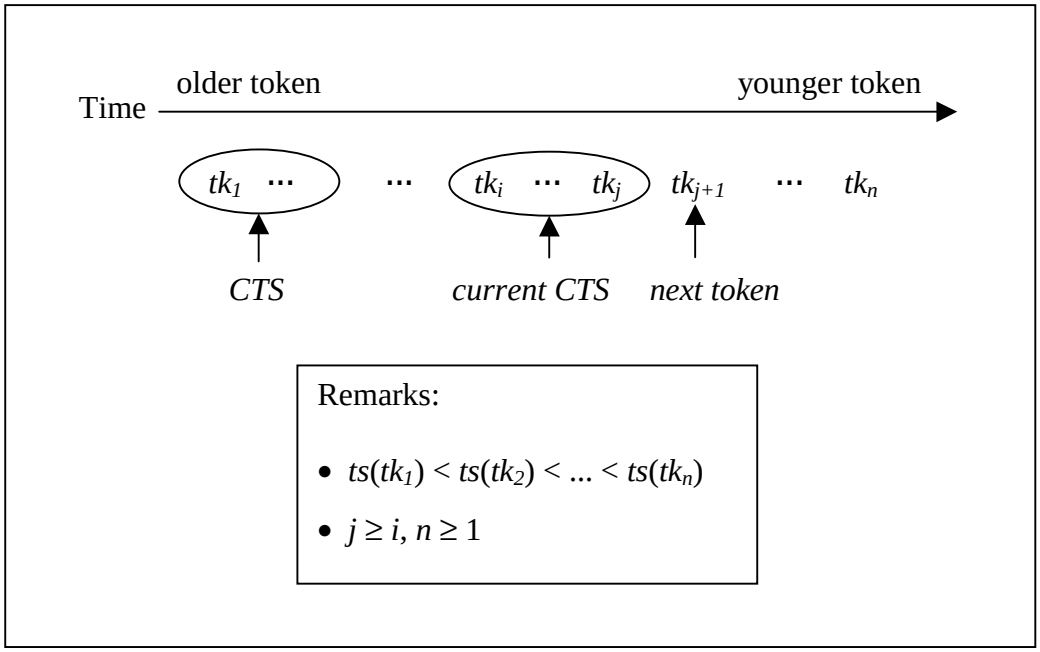


Figure 7.9: Concurrent token set detection

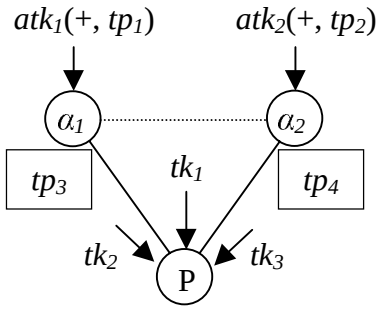
The following subsections explain CTS detection methods for a trigger without event clause (Subsection 7.4.1.1), for a trigger with event clause (Subsection 7.4.1.2).

7.4.1.1 CTS detection for a trigger without event clause

Basically, we assume that the tokens are delivered in the timestamp order and that there is a primary key constraint on the tables that are the data sources of the triggers. Based on these assumptions, the consecutive + tokens that arrive at the same α node belong to different token families. Therefore, the tokens can be processed in parallel

since this changes only the order of rule action execution. Similarly, the consecutive – tokens that arrive at the same α node belong to different token families and can be processed in parallel.

An example is shown in Figure 7.10, where atk_1 arrives at α_1 and atk_2 arrives at α_2. Assume tp_1 joins with tp_2, tp_1 joins with tp_4, and tp_2 joins with tp_3. Any order of processing of atk_1 and atk_2 will create compound tokens $tk_1(+, tp_1, tp_2)$, $tk_2(+, tp_1, tp_4)$, and $tk_3(+, tp_3, tp_2)$, which means atk_1 and atk_2 can be processed in parallel. Therefore, the consecutive + tokens that arrive at the different α nodes can be processed in parallel. In this writing, the *different α nodes* mean the *different α nodes in the same discrimination network.* Similarly, the consecutive – tokens that arrive at the different α nodes can be processed in parallel.



Remarks:

- join(tp_1,tp_2)
- join(tp_1,tp_4)
- join(tp_3,tp_2)
- tk_1 (+, tp_1, tp_2)
- tk_2 (+, tp_1, tp_4)
- tk_3 (+, tp_3, tp_2)

Figure 7.10: A Gator network for a trigger without event clause

If the tokens that arrive at the same α node have different event types, then the parallel processing of them can corrupt the α node. If the tokens that arrive at the same

discrimination network have different event types, then the parallel processing of them can cause an untimely joining error. The CTS detection policy can be summarized as:

- A group of consecutive + tokens that arrive at different α nodes can be processed in parallel.
- A group of consecutive – tokens that arrive at different α nodes can be processed in parallel.
- Other token combinations must be processed sequentially.

```
procedure determine_inclusion_one (tk1, CTS1)
   // Since each CTS has tokens with the same event type (+, –, or δ), we call the
   // event type of the tokens in a CTS the event type of the CTS.
   (1) evtCTS1 ← event type of CTS1
   (2) evttk1 ← event type of tk1
   (3) if evtCTS1 = 'δ' then return false endif
   (4) case
   (5)      : evttk1 = '+' : if evtCTS1 = '+' then return true else return false endif
   (6)      : evttk1 = '–' : if evtCTS1 = '–' then return true else return false endif
   (7)      : evttk1 = 'δ' : return false
   (8) endcase
end determine_inclusion_one
```

Figure 7.11: Inclusion test for a token into a CTS (without event clause)

The algorithm for determining the inclusion of the next token into a CTS is described in the procedure *determine_inclusion_one* shown in Figure 7.11. It is not difficult to see that only constant time is needed to determine whether the next token could be included into the CTS.

An example is given in Figure 7.12, where ten tokens arrive in ascending order of the token subscripts at a Gator network with three α nodes α_1, α_2, and α_3. The tokens, tk_3, tk_4, and tk_5 are + tokens and can be processed in parallel. They constitute a CTS $\{tk_3, tk_4, tk_5\}$.

Remarks:

- $tk_1[+:\alpha_1]$: tk_1 is an insert token that arrives at α_1.
- $ts(tk_1) < ts(tk_2) < ts(tk_3) < \cdots < ts(tk_{10})$
- $\rightarrow$: token arrival order (: followed by)
- ➔ : CTS processing order (: followed by)

Ten tokens arriving at three α nodes

$tk_1 \rightarrow tk_2 \rightarrow tk_3 \rightarrow tk_4 \rightarrow tk_5 \rightarrow tk_6 \rightarrow tk_7 \rightarrow tk_8 \rightarrow tk_9 \rightarrow tk_{10}$

$[+:\alpha_1]$ $[-:\alpha_3]$ $[+:\alpha_1]$ $[+:\alpha_2]$ $[+:\alpha_2]$ $[-:\alpha_2]$ $[-:\alpha_1]$ $[-:\alpha_3]$ $[\delta:\alpha_1]$ $[\delta:\alpha_3]$

Detected CTSs

$\{tk_1\}$➔ $\{tk_2\}$➔ $\{tk_3, tk_4, tk_5\}$➔ $\{tk_6, tk_7, tk_8\}$➔ $\{tk_9\}$➔ $\{tk_{10}\}$

Figure 7.12: Tokens and CTSs

7.4.1.2 CTS detection for a trigger with event clause

An α node that has an event clause on it is called a ***node with event clause***. For example, α_3 in Figure 7.13 is a *node with event clause*. A token that arrives at a *node with event clause* and has the same event type as the event clause of the node is called an ***event-qualifying token***. For example, atk_3 in Figure 7.13 is an *event-qualifying token*.

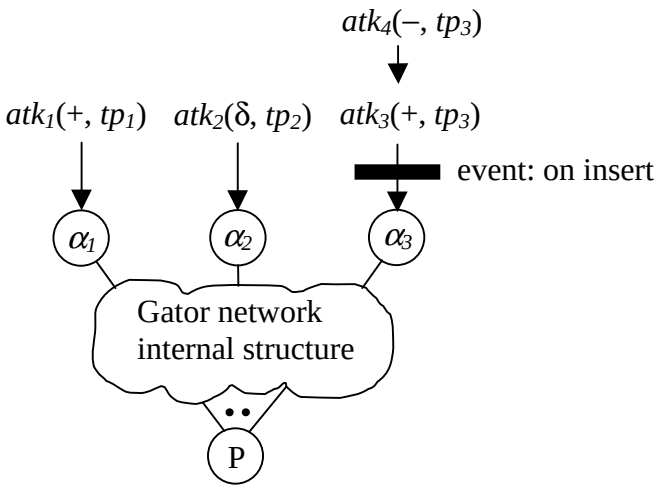


Figure 7.13: A Gator network for a trigger with insert event clause

The CTS detection **policy** can be summarized as:

(1) An *event-qualifying token* can be processed in parallel only with other *event-qualifying token*s.

(2) A non-*event-qualifying token* can be processed in parallel with other non-*event-qualifying token*s that arrive at different α nodes. For example, in Figure 7.13, atk_1, atk_2, and atk_4 can be processed in parallel.

(3) A + token can be processed in parallel with other + tokens that arrive at the same α node.

(4) A – token can be processed in parallel with other – tokens that arrive at the same α node.

(5) Other token pairs cannot be processed in parallel. For example, the processing of atk_3 and atk_4 can corrupt the content of α_3, and the parallel processing of atk_1 and atk_3 can create an *untimely joining error*.

To determine the inclusion of the next token into the current CTS in a constant time, the following flags are needed for a discrimination network, *N*.

- f_{evt} (The flag for the current CTS): *True* when one or more event-qualifying tokens are in the current CTS, *false* otherwise.
- The flags for each α node α_i of *N*:
  - $f_{pls}(\alpha_i)$: *True* when a + token that arrives at α_i is in the current CTS, *false* otherwise.
  - $f_{mns}(\alpha_i)$: *True* when a – token that arrives at α_i is in the current CTS, *false* otherwise.
  - $f_{dlta}(\alpha_i)$: *True* when a δ token that arrives at α_i is in the current CTS, *false* otherwise.

These flags will be cleared to *false* at the end of each CTS detection. The algorithm that determines the inclusion of the next token into the current CTS is given in the procedure *determine_inclusion_two* (Figure 7.14). We can see that the execution of the procedure requires only constant time.

```
procedure determine_inclusion_two (tk2, α2, CTS2)
  // tk2: a token that arrives at an α node α2 of a discrimination network N
  // CTS2: the current CTS for N
  // fevt: a flag that specifies whether event-qualifying tokens are in CTS2
  // fpls(α2), fmns(α2), fdlta(α2): flags that specify the existence of a +/–/δ token that
  //                                arrive at α2 in the current CTS
  // Policy 3 is passively implemented in Line 9 by not checking flag fpls.
  // Policy 4 is passively implemented in Line 11 by not checking flag fmns.
  (1)  evtCTS2 ← event type of CTS2
  (2)  evttk2 ← event type of tk2
  (3)  if tk2 is an event_qualifying token then
  (4)       if fevt then return true // policy (1)
  (5)       else if CTS2 is empty then fevt ← true; return true
  (6)       else return false
  (7)       endif endif
  (8)  else if fevt then return false // policy (1)
  (9)  else case : evttk2 = '+' : if fmns(α2) or fdlta(α2) then return false // policy (3), (5)
(10)                             else fpls(α2) ← true; return true endif // policy (2)
(11)               : evttk2 = '–' : if fpls(α2) or fdlta(α2) then return false // policy (4), (5)
(12)                             else fmns(α2) ← true; return true endif // policy (2)
(13)               : evttk2 = 'δ' : if fpls(α2) or fmns(α2) or fdlta(α2)
(14)                                                 then return false // policy (5)
(15)                             else fdlta(α2) ← true; return true endif // policy (2)
(16)       endcase
(17)  endif endif
end determine_inclusion_two
```

Figure 7.14: Inclusion test for a token into a CTS (with event clause)

As an example of CTS detection for the triggers with an insert event clause, consider the ten tokens of Figure 7.12 and assume that they arrive at the Gator network in Figure 7.13. For example, the non-*event-qualifying token* $tk_5[+:\alpha_2]$ can be processed in parallel with $tk_1[+:\alpha_1]$, $tk_2[-:\alpha_3]$, $tk_3[+:\alpha_1]$, and $tk_4[+:\alpha_2]$ according to the policy (2), (2), (2), and (3), respectively. After all, the CTSs would be:

$$\{tk_1, tk_2, tk_3, tk_4, tk_5\} \rightarrow \{tk_6, tk_7, tk_8\} \rightarrow \{tk_9, tk_{10}\}$$

For the triggers with delete event clause, assume a Gator network G_{del} that is produced by modifying the Gator network in Figure 7.13. In G_{del}, α_3 is still a *node with event clause*, but α_3 has a delete event clause, instead of an insert event clause, on it. Further assume that the ten tokens of Figure 7.12 arrive at G_{del}. In this case, the *event-qualifying token* $tk_2[-:\alpha_3]$ cannot be processed with the non-*event-qualifying token*s $tk_1[+:\alpha_1]$ and $tk_3[+:\alpha_1]$. After all this, the CTSs would be:

$$\{tk_1\} \rightarrow \{tk_2\} \rightarrow \{tk_3, tk_4, tk_5\} \rightarrow \{tk_6, tk_7\} \rightarrow \{tk_8\} \rightarrow \{tk_9, tk_{10}\}$$

For triggers with an update event clause, assume a Gator network G_{upd} that is produced by modifying the Gator network in Figure 7.13. In G_{upd}, α_3 is again a *node with event clause*, but α_3 has an update event clause, instead of an insert event clause, on it. Further assume that the ten tokens of Figure 7.12 arrive at G_{upd}. In this situation, the token $tk_9[\delta:\alpha_1]$ cannot be processed in parallel with $tk_7[-:\alpha_1]$ and tk_9 cannot be processed in parallel with an *event-qualifying token* $tk_{10}[\delta:\alpha_3]$. After all this, the CTSs would be:

$$\{tk_1, tk_2, tk_3, tk_4, tk_5\} \rightarrow \{tk_6, tk_7, tk_8\} \rightarrow \{tk_9\} \rightarrow \{tk_{10}\}$$

7.4.2 Parallel Token Processing Architecture

Suppose *n* tokens are in the token (arrival) queue (AQ in Figure 7.15), and the number of triggers defined in the system is *m*. In the worst case, each token could be included in a CTS for the discrimination network of every trigger. Therefore, the worst case storage requirement for all the CTSs of every discrimination networks is $O(mn)$. Hence, it is inefficient detecting all the CTSs for each discrimination network and then processing them, because of the high storage requirement. Additionally, such stepwise processing unnecessarily delays the execution of the actions of triggers with satisfied conditions.

In this section, we propose a token processing architecture that reduces the storage requirement and initiates the processing of the tokens as soon as they arrive at the system. The proposed architecture consists of three kinds of token queues and two kinds of processes. The architecture is depicted in Figure 7.15. The token queues include one instance of the *Arrival Queue* (AQ) and the *Ready Queue* (RQ) and one instance of the *Waiting Queue* (WQ) per discrimination network in the system. The processes in the system include the *CTS detecting process* and the *token handling process*.

The AQ stores the tokens that are delivered from the data sources on which one or more triggers are defined. Only one CTS for a discrimination network can be executed at one moment. Therefore, a maximum of one CTS for a discrimination network can be in the RQ. Hence, a WQ is needed per discrimination network, N_1, to store the extra CTSs of N_1. For notational convenience, we will use RQ CTS to imply a CTS in the RQ.

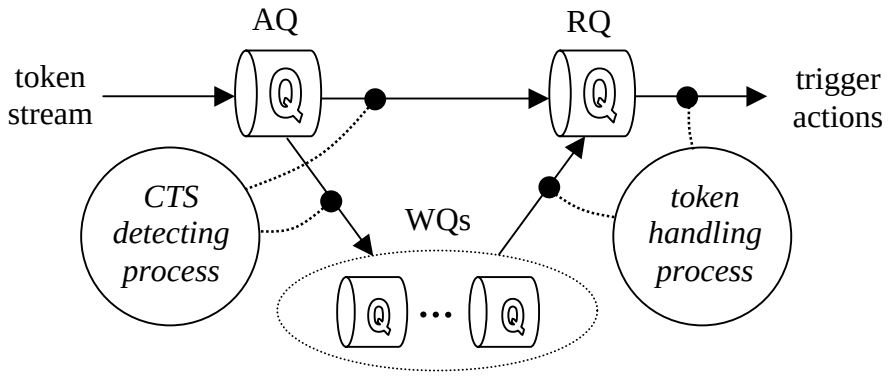


Figure 7.15: The three-token-queue architecture

Basically, the *CTS detecting process* consumes the tokens in the AQ and sends them to the RQ or the WQs. To limit the storage requirement of the RQ and the WQs, the concept of maximum *threshold* on the size of RQ is used. If the RQ is shorter than the *threshold*, then the *CTS detecting process* dequeues a token from the AQ, and sends it through the Selection Predicate Index structure [40]. The *CTS detecting process* identifies α nodes (across all discrimination networks) at which the tokens arrive. For each identified discrimination network *N* that is reached by a token, *tk*, the *CTS detecting process* either includes *tk* into the last CTS for *N* or creates a new CTS for *N*. The last CTS for *N* can be either in the WQ of *N* or in the RQ.

If the threshold were set too high, then the WQs and the RQ would require excessive storage. If the threshold were set too low, then the benefit of parallel processing would be reduced. Therefore, setting the threshold for the RQ length is an optimization problem.

```
procedure CTS_detecting_process
 // threshold: a system parameter that limits the maximum length of RQ
 (1)  while true do
 (2)      if (length (RQ) ≥ threshold)
 (3)          then wait until RQ becomes shorter than threshold endif
 (4)      if (length (AQ) = 0)
 (5)          then wait until AQ has one or more tokens endif
 (6)      tk ← dequeue (AQ)
 (7)      send tk through the SPI structure // Find α nodes (of all discrimination
                                            // networks) at which tk arrives.
 (8)      for each α node αi at which tk arrives do // Let N be the discrimination
                                                    // network that contains αi.
 (9)          if length (the WQ for N) = 0 then
(10)              if a RQ CTS for N (RQ_CTS_N) exists then
(11)                  if tk can be included in RQ_CTS_N then // Section 7.4.1
(12)                      put (tk, N, αi) into RQ_CTS_N // direct insertion
(13)                  else initialize a CTS with (tk, N, αi)
(14)                      enqueue the CTS into the WQ for N endif
(15)              else initialize a CTS with (tk, N, αi)
(16)                  enqueue the CTS into RQ endif // direct insertion
(17)          else if tk can be included in the last WQ CTS for N then // Section 7.4.1
(18)               put (tk, N, αi) into the last WQ CTS
(19)          else initialize a CTS with (tk, N, αi)
(20)              enqueue the CTS into the WQ for N
(21)          endif endif
(22)  repeat repeat
end CTS_detecting_process
```

Figure 7.16: Procedure *CTS_detecting_process*

Assume the WQ for a discrimination network *N* is empty and a token, *tk*, arrives at *N*. If the RQ CTS for *N* is empty or *tk* can be processed in parallel with each token in the RQ CTS for *N*, then the *CTS detecting process* will insert *tk* directly into the RQ CTS for *N*. We will call this a ***direct insertion***. By doing this, the processing of *tk* could start as early as possible. In fact, it could be inefficient to delay the processing of *tk* until the CTS that contains *tk* is detected completely. The algorithm for the *CTS detecting process* is given in the procedure *CTS_ detecting_process* (Figure 7.16).

The *token handling process* consumes the tokens in the RQ CTSs and relocates the CTSs from the WQs to the RQ. The *process* deletes a token *tk* from the RQ CTS for a discrimination network, *N*, as soon as *tk* is processed. We call this an ***immediate deletion***. By doing this, the chance of inclusion of the next token arriving at *N* into the RQ CTS for *N* will be increased when the WQ CTS for *N* is empty. Therefore, the overall performance of the system could be increased. Additionally, the time needed to determine whether a token could be included in a CTS will be reduced.

An example in Figure 7.17 shows how the *direct insertion* and the *immediate deletion* affect the CTS detection and increase the system performance. In the figure, tk_1, tk_2, and tk_3 arrive at the Gator network *G* consecutively. When tk_1 and tk_2 in the first CTS are processed in parallel, if tk_3 arrived after the processing of tk_1 is done and before the processing of tk_2 is done, then the processing of tk_3 can be overlapped with the processing of tk_2, since tk_3 can be processed in parallel with tk_2. The *immediate deletion* and the *direct insertion* made the parallel processing of tk_2 and tk_3 possible. In summary, the *immediate deletion* combined with the *direct insertion* could increase the number of tokens that can be processed in parallel.

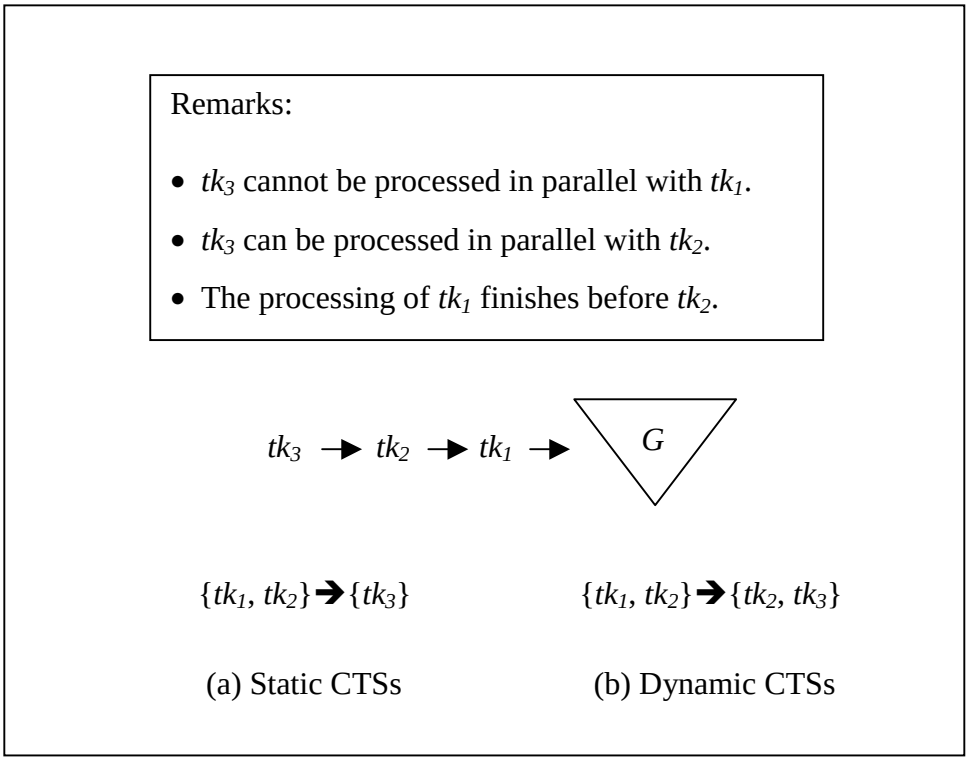


Figure 7.17: The effect of the *direct insertion* and the *immediate deletion*

After processing all the tokens in the RQ CTS for a discrimination network *N*, if the WQ of *N* is not empty, then the *token handling process* relocates the first WQ CTS for *N* to the RQ. The algorithm for the *process* is given in the procedure *token_handling_process* (Figure 7.18).

7.4.3 Summary of Level 2 Consistency

Our solution to achieve the Level 2 consistency includes the detection of the *concurrent token set*s (CTSs) from among the tokens that arrive at a discrimination network and the architecture for the parallel token processing. The concept of the CTS was developed to increase the performance of the system through the parallel processing

of the tokens. A CTS is a set of consecutive tokens that arrive at a discrimination network such that the parallel processing of them provides Level 2 consistency.

```
procedure token_handling_process
 (1)  while true do
 (2)      if length (RQ) = 0
 (3)          then wait until RQ is not empty endif
 (4)      (tk, N, αi) ← next item from RQ that is not yet being processed
 (5)      process tk against αi and propatage tk
 (6)      delete (tk, N, αi) from RQ // really delete the item -- immediate deletion
 (7)      if length (RQ CTS for N) = 0 then
 (8)          if length (WQ for N) ≠ 0 then
 (9)              CST_N ← dequeue (WQ for N); enqueue CST_N into RQ endif
(10)      endif
(11)  repeat
end token_handling_process
```

Figure 7.18: Procedure *token_handling_process*

The CTS detection is based on the event types (insert, delete, or update) of the tokens, the α nodes at which the tokens arrive, and whether the α nodes are the *node with event clause*. Provided in this section (Section 7.4) are the CTS detection policies and the algorithms that determine whether a token can be included into a CTS for the triggers with or without event clause. We showed that the time needed to check whether a token can be included in a CTS of size *n* is a constant.

The proposed architecture for the parallel token processing consists of three kinds of token queues and two kinds of processes. The token queues include one instance of

the *Arrival Queue* and the *Ready Queue* and one instance of the *Waiting Queue* per discrimination network in the system. The processes in the system include the *CTS detecting process* and the *token handling process*. The techniques that are employed by these processes are the *direct insertion* and the *immediate deletion*. The benefits of these techniques include the increase of the number of tokens that can be processed in parallel, and the reduction of the time that is needed to determine whether a token can be included in a CTS.

A virtual α node inherently has the problem of *untimely joining error*. To prevent the untimely joining error while supporting virtual α nodes, *shadow tables* need to be employed for the virtual α nodes to provide the Level 2 consistency.

The value of our solution depends on the token delivery pattern and the type of the triggers defined in the system. As an example, assume a trigger without event clause is defined on two data sources ds_1 and ds_2. If ds_1 and ds_2 repeatedly send 1000 consecutive + tokens in turn, then the 1000 tokens can form a CTS and can be processed in parallel. Without our solution, the 1000 tokens need to be processed in serial to provide Level 2 consistency. In this case, our solution will provide a linear speedup. We believe that an insert-mostly workload is the common case, so a high speedup can often be achieved.

From the fact that the tokens that arrive at different discrimination networks can be processed in parallel, we can think of a different situation. When many triggers are defined in the system, the parallel processing of the tokens that reach different discrimination networks can produce enough workload to keep each processor in the system busy all the time. In this case, the cost of our solution could exceed the benefit.

7.5 Trigger Processing Consistency Level 1

The conditions for Level 1 consistency include:

(i) The contents of the memory nodes of the discrimination network converge.

(ii) The *timing error* in the joining tuples is limited to some fixed value.

Level 1 consistency allows the *untimely joining error* to occur and processes every token that arrives at the same trigger in parallel. The tokens that arrive at different triggers can always be processed in parallel. As a result, all tokens that arrive at the system during a cycle can be processed in parallel and still provide Level 1 consistency. No CTS detection (Subsection 7.4.1) is needed to provide Level 1 consistency. Therefore, the reduced computation unit is used for Level 1 consistency.

However, simple concurrent token processing can corrupt the contents of memory nodes of a discrimination network and cannot make condition (i) hold. Therefore, we propose a special mechanism called ***Stability-Lookaside Buffer*** (SLB) to make the memory nodes converge. In this writing, *converge* and *stabilize* are equivalent. Condition (ii) could be guaranteed by processing only the tokens that arrive during a *cycle* in parallel. The *timing error* is limited to the time period during which the tokens that form a batch job are accumulated.

This section is organized as follows: Subsections 7.5.1 and 7.5.2 introduce the stabilization techniques for α nodes and β nodes, respectively, Subsection 7.5.3 explains the necessity of a *shadow table* for a virtual α node, and Subsection 7.5.4 summarizes Level 1 consistency.

7.5.1 Stabilization of α Nodes

The concurrent processing of tokens that arrive at an α node can corrupt the α node. For example, if a tuple is inserted into and then deleted from the base table of an α node during a short period of time, then two tokens (one + token and one − token) could be delivered and exist in the token queue simultaneously. When the two tokens are processed concurrently, if the − token is processed later, then there will be no problem. However, if the + token is processed after the − token, then there will be a tuple in the memory node but it should have been previously deleted.

Therefore, we propose a technique that protects α nodes from being corrupted during the concurrent processing of tokens that arrive at the nodes. In the computer hardware arena, a Translation-Lookaside Buffer (TLB) keeps the recent address translations and increases the address translation speed in the virtual memory system [65]. The concept of the TLB is adopted in TriggerMan to stabilize the memory node. The adopted concept will be called an SLB.

However, the SLB of TriggerMan is quite different from the TLB in terms of theoretical background (locality: TLB vs. anti-locality: SLB), content (address translation: TLB vs. tuple modification status: SLB), and purpose (increase translation speed: TLB vs. stabilize node content: SLB). The SLB that stores atomic tokens is called an ***atomic* SLB** and the SLB that stores compound tokens is called a ***compound* SLB**.

To refresh the readers' memory, let us repeat the definition of *family* here. A ***family*** is a set of objects that have the same key and are related with the same node of a

discrimination network. The objects can be a token, a tuple, or an SLB *line* (Subsection 7.5.1.1). The whole technique to make α nodes stabilize is called ***Strategy I***.

7.5.1.1 Strategy I

To explain Strategy I, we need to describe the content of an SLB for an α node, the operation of the SLB, and the processing of the tokens that arrive at an α node. To make α nodes stabilize an atomic SLB is allocated to an α node. The SLB will contain three pieces of information (<key, event type, timestamp>) for each tuple modified (inserted/updated/deleted) during the current cycle. This information is called a ***line***. Each *line* contains the information about the token that has been applied to a tuple most recently.

Each token has an event type. There are three kinds of event types: +, –, or δ. An insert, a delete, or an update operation, respectively, against a base table generates these event types. We assume that a token with event type δ (a δ token) is processed as a – token followed by a + token. The usual processing of a + or a – token, *tk*, that arrives at a node *n* is given below. (For more details on token processing refer to [34] and [36].) When *tk* is a:

+ token: Insert *tk* into *n*. If *n* has one or more siblings, then make compound tuples using matching tuples in the siblings and create compound tokens out of the compound tuples, and propagate them to the parent node of *n*.

– token: Delete *tk* from *n*. The *tk* can be propagated to the parent nodes of *n* in one of two forms -- either an atomic token or a compound token. When *tk* is propagated in the form of an atomic token, each node at which *tk*

arrives needs to be scanned to determine if the node contains tuples that have *tk* as a component. If so, the tuples are deleted.

Table 7.2: The processing of the first atomic token in a family

case	current token event type	tuple exists in the node	action (In each case, create a line in SLB using the current token)
1	+	yes	Delete the α node tuple. Create a – token for the deleted tuple and propagate it to the parent nodes. Process the current token as usual.
2		no	Process the current token as usual.
3	–	yes	Process the current token as usual.
4		no	Do not process the current token since there is no token to delete.
5	δ	yes	Repeat the action given in case 1*.
6		no	Process current token as a + token.

* Due to concurrent processing, we do not know whether the update (δ) operation can be applied to the tuple in the node. Therefore, a δ token needs to be replaced by a – token which is then followed by a + token.

The operations of an SLB are as follows:

- An SLB is cleared after each cycle.
- If the SLB does not have a line that is in the same family as the token under consideration (current token), then it (the first token in a family) is processed according to Table 7.2.

- If the SLB has a line that is in the same family as the current token;
  - When the line is younger than the current token, the current token is discarded.
  - When the current token is younger than the line, then the token (the second or subsequent token in a family) is processed according to Table 7.3.

Table 7.3: The processing of the second or subsequent atomic token in a family

case	token event type		action (In each case, update the line in SLB using the current token)
	current	in SLB	
1	+	+	Delete the α node tuple. Create a – token for the deleted tuple and propagate it to the parent nodes. Process the current token as usual.
2	+	–	Process the current token as usual.
3	+	δ	Repeat the action given in case 1.
4	–	+/δ	Process the current token as usual.
5	–	–	Do not process the current token.
6	δ	+/δ	Repeat the action given in case 1*.
7	δ	–	Process current token as an insert token.

* Refer to the explanation in Table 7.2.

7.5.1.2 Proof of α node stabilization

A token *tk* is said to ***influence*** a tuple *tp* when *tk* determines the existence and content of *tp*. Using the term *influence*, the stabilization of an α node is proven by Theorem 7.1.

Theorem 7.1: If the tokens that arrive at an α node are processed using *Strategy I*, then the α 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 at the end of each cycle. Each tuple in an α node is associated with a family of tokens in each cycle. Let *F* be a set of tokens in a *family* that is associated with an α node tuple tp_1 in a cycle c_1. We can prove the hypothesis by establishing that the youngest token in *F influences* tp_1 at the end of c_1.

If the number of tokens in *F* (the size of *F*) is 0 or 1, then obviously tp_1 stabilizes at the end of c_1. If the size of *F* is greater than 1, then, since each token in *F* will be processed eventually, the influence of the youngest token to tp_1 will be established by proving that the youngest token among the processed tokens in *F influences* tp_1. This will be proved using mathematical induction on the number of processed tokens in *F*.

Basis: When the number of processed tokens in *F* is 1, let tk_1 be the processed token. Then, tk_1 is the youngest token since it is unique and will certainly *influence* tp_1 by Table 7.2.

Induction: Suppose that the hypothesis is true when the number of processed tokens in *F* is *m*. Let tk_2 be in *F* and processed in (*m*+1)-th order. Let tk_m be the youngest token among the *m* processed tokens. Then, by the induction hypothesis, tk_m influences tp_1 just

before tk_2 is processed. If tk_2 is younger than tk_m, then tk_2 is the youngest among the m+1 tokens and will certainly *influence* tp_1 by Table 7.3. Otherwise (if tk_2 is older than tk_m), then tk_2 will be discarded by Strategy I. In this case, tk_m is the youngest token among the m+1 tokens and will continuously *influence* tp_1. In both cases, the youngest token among the m+1 tokens *influences* tp_1. Therefore, the stabilization of tp_1 is established, and it proves the stabilization of an α node. □

7.5.2 Stabilization of β Nodes

In this subsection organized as follows: the necessity of the dummy timestamp 0 is explained in Subsection 7.5.2.1, the proposed strategy, ***Strategy II***, for the β node stabilization is given in Subsection 7.5.2.2, and the proof of the stabilization of a β node in Strategy II appears in Subsection 7.5.2.3.

7.5.2.1 The necessity of the dummy timestamp 0

To make the timestamp vector of a compound token complete, a dummy timestamp needs to be assigned to a compound token component that is not modified in the current cycle. The necessity of a dummy timestamp is illustrated in Figure 7.19.

Originally, three atomic tokens, atk_1, atk_2, and atk_3, arrive at the Gator network in the figure. The situation after atk_1 and atk_3 are processed consecutively is shown in Figure 7.19. They are propagated to P in the form of tk_1 and tk_3. Assume that atk_2 is discarded by Strategy I. Two compound tokens, tk_1 and tk_3, are to be processed in parallel. Among tk_1 and tk_3, whichever is processed first will be stored in the SLB of P. When the second processed token arrives, the timestamps of the token and the SLB line

need to be compared to determine the destiny of the second processed token. Therefore, a dummy timestamp is needed for the first component of tk_1.

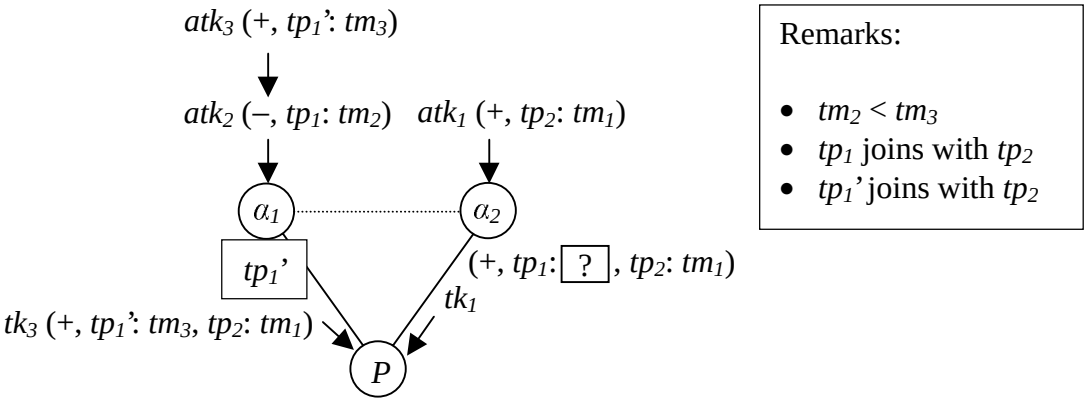


Figure 7.19: The necessity of dummy timestamp 0

In this case tk_1 needs to be evaluated to be older than tk_3. That is, the dummy timestamp that will be assigned to tk_1 needs to be less than (older than) any real timestamp. Since 0 is the lowest (oldest) timestamp, 0 is used as the dummy timestamp for the compound token component that is not modified in the current cycle.

7.5.2.2 Strategy II

Strategy II is developed based on Strategy I that is for α node stabilization. In Strategy II, when a – token arrives at a memory node *n*, TriggerMan will join the token with the sibling node(s) of *n* and propagate the results to the parent nodes of *n* in the form of – compound token(s). Therefore, each token that arrives at a β node is a compound

token. By doing this, the SLB structure for a β node can be simplified. That is, to make β nodes stabilize a compound SLB is enough for each β node.

To explain Strategy II, we need to describe the assignment of timestamps to a compound token, the content and the manipulation of the SLB, and the processing of a compound token. These issues are covered in the following subsections.

7.5.2.2.1 Timestamp assignment to a compound token

The timestamps of a compound token are derived from the timestamps of the components of the token. Hence, a compound token with *m* component has *m* timestamps. The timestamp vector of a compound token is used in determining the younger token between two compound tokens.

When a token (either a + token or a − token) arrives at a node *n*, it will be propagated to the parent nodes of *n* after joining the token with the siblings of *n*. Let *tk* be a compound token that arrives at a β node. Then, timestamps are assigned to the components of *tk* as follows:

- If *c* has a line, *l*, in the SLB of an α node from which *c* come, then the timestamp of *l* is assigned to *c*.
- Otherwise (if *c* does not have a line in the α node SLB), 0 is assigned to *c*.

An example is given in Figure 7.20, where the SLB of α_2 shows that tp_2 has been inserted in α_2 in the current cycle. Since the SLBs are empty, tp_3 and tp_4 were inserted into α_3 and α_4, respectively, in some previous cycles. The atomic token atk_1 arrives at α_1

after a compound tuple, (tp_2, tp_3), has been inserted into β_1. Note the timestamps of tp_3 and tp_4 in tk_1 are 0.

7.5.2.2.2 Content and manipulation of SLB

Each β node is assigned a compound SLB. The content of each line in the compound SLB of a β node will be:

- The event type (+/–) of the token.
- For each component of the token, the key of the component and the timestamp associated with the component.

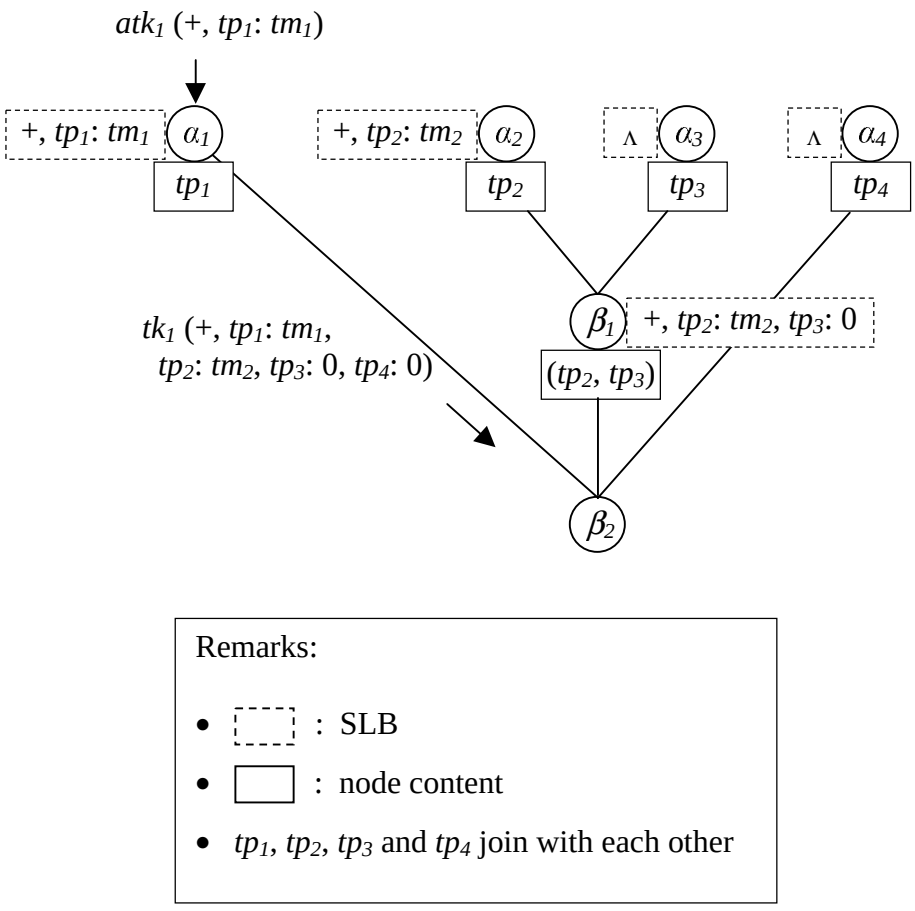


Figure 7.20: Assigning timestamps to a compound token.

An example of an SLB of a β node is given in the SLB of β_1 in Figure 7.20.

7.5.2.2.3 Token processing policy

The SLB of a β node is cleared at the end of each cycle. A token, *tk*, that arrives at a β node, β_1, is processed as follows:

- If the SLB of β_1 does not have a line that is in the same family as *tk*, then *tk* is processed using Table 7.4.
- Otherwise (the SLB of β_1 contains a line, *l*, that is in the same family as *tk*):
  - If *tk* is younger than *l*, then *tk* is processed using Table 7.5.
  - Otherwise (*tk* is older than *l*), *tk* is discarded.

Table 7.4: The processing of the first compound token in a family

case	current token event type	tuple exists in the node	action (in each case, create a line in SLB using the current token)
1	+	yes	Create a compensating – token and apply it to the β node and propagate it to the parent nodes. Process the current token as usual.
2		no	Process the current token as usual.
3	–	yes	Process the current token as usual.
4		no	Do not process the current token since there is no token to delete.

The event type of a token that arrives at a β node is either + or –. This is because the δ token that arrives at an α node is replaced by a – token which is then followed by a + token.

Table 7.5: The processing of the second or subsequent compound token in a family

case	token event type		action (in each case, update the line in SLB using the current token)
	current	in SLB	
1	+	+	Create a compensating – token and apply it to the β node and propagate it to the parent nodes. Process the current token as usual.
2		–	Process the current token as usual.
3	–	+	Process the current token as usual.
4		–	Do not process the current token.

7.5.2.3 Proof of β node stabilization in Strategy II

The proof of β node stabilization is embodied in Lemma 7.1, Lemma 7.2, and Theorem 7.2.

Lemma 7.1: If the compound tokens are generated using Strategy II, then each compound token in a *family* that arrives at a β node has a unique timestamp vector.

Proof: We will use a contradiction in proving the hypothesis. Let tk_1 and tk_2 be compound tokens in the same family that arrives at a β node. Assume that tk_1 and tk_2 have the same timestamp vector. Let itk_1 and itk_2 be the initiating tokens of tk_1 and tk_2,

respectively. Since tk_1 and tk_2 have the same timestamp vector, itk_1 and itk_2 both are components of tk_1 and of tk_2.

If itk_1 and itk_2 are from the same α node, then it means the same atomic token arrived at the α node more than once. This is impossible in normal operations. Therefore, itk_1 and itk_2 are from different α nodes. Let cu_1 and cu_2 be the *computation units* that process itk_1 and itk_2, respectively. If cu_1 was processed first, then tk_1 could not include itk_2. If cu_2 was processed first, then tk_2 could not include itk_1. Therefore, any execution sequence of cu_1 and cu_2 under correct concurrency control cannot produce tk_1 and tk_2 together. This result would contradict the assumption. Hence, each compound token in a *family* that arrives at a β node has a unique timestamp vector. □

Lemma 7.2: If the two compound tokens that are generated using Strategy II belong to the same *family*, then they are *comparable*.

Proof: We will use a contradiction in proving the hypothesis. Let tk_1 and tk_2 be compound tokens that are generated using Strategy II; they arrive at a β node, β_1, and belong to the same *family*. Assume that tk_1 and tk_2 are *incomparable*. Then, since tk_1 and tk_2 have different timestamp vectors by Lemma 7.1, components c_{11} and c_{12} exist in tk_1 and components c_{21} and c_{22} exist in tk_2 and satisfy the conditions:

- c_{11} is in the same position as c_{21}, and c_{12} is in the same position as c_{22}
- $ts(c_{11}) > ts(c_{21})$, and $ts(c_{12}) < ts(c_{22})$

Let cu_1 and cu_2 be the *computation units* that produce tk_1 and tk_2, respectively. If cu_1 is processed first, then c_{11} must be stored in the associated α node α_1, the parent node p_1 of α_1, the parent node p_2 of p_1, and so on, up to a child node of β_1. Later, when cu_2 is

processed it cannot see c_{21} that is older than c_{11}. This is true since the timestamp of a component in a memory node cannot become older. Therefore, cu_2 cannot create tk_2. Because of the same reasoning, when cu_2 is processed first, cu_1 cannot create tk_1.

Any execution sequence of cu_1 and cu_2 under correct concurrency control cannot produce tk_1 and tk_2 together. This result would contradict the assumption. Hence, tk_1 and tk_2 are comparable and the theorem is proven. □

Theorem 7.2: If the tokens that arrive at a β node are processed using *Strategy II*, then the β node stabilizes at the end of each cycle.

Proof: Since a β node is a set of tuples, a β node stabilizes if and only if each tuple in the node stabilizes at the end of each cycle. Each tuple in a β node is associated with a family of tokens in each cycle. Let F be the set of tokens in a family that is associated with a β node tuple tp_1 in a cycle c_1. We can prove the hypothesis by establishing that the youngest token in F *influences* tp_1 at the end of c_1.

If the number of tokens in F (the size of F) is 0 or 1, then tp_1 will obviously stabilize at the end of c_1. If the size of F is greater than 1, then we will prove the influence of the youngest token in F, using mathematical induction on the number of processed tokens in F.

Basis: When the number of processed tokens in F is 1, let tk_1 be the processed token. Then, tk_1 is the youngest token since it is unique and will certainly *influence* tp_1 by Table 7.4.

Induction: Suppose that the hypothesis is true when the number of processed tokens in F is m. Let tk_2 be the token in F and processed in (m+1)-th order. Let tk_m be the youngest

token among the *m* tokens that have been processed. By the induction hypothesis, tk_m *influences* tp_1 just before tk_2 is processed. By Lemma 7.2, tk_2 is *comparable* with tk_m. If tk_2 is younger than tk_m, then tk_2 is the youngest token among the m+1 tokens and will certainly influence tp_1 by Table 7.5. Otherwise (if tk_2 is older than tk_m), tk_2 will be discarded by Strategy II. In this case, tk_m is the youngest token among the m+1 tokens and will continuously *influence* tp_1. In both cases, the youngest token among m+1 tokens *influences* tp_1. Therefore, the stabilization of tp_1 is established, and it proves the stabilization of a β node. □

7.5.3 Necessity of Shadow Table for Virtual α Nodes

When the tokens are delivered from the data sources, TriggerMan accumulates them and creates a batch. A ***cycle*** is the time during which the tokens of one batch are processed. The length of a cycle depends on the amount of work in a batch (*batch size*). If the length of the cycle is greater than the period during which the batch is formulated, then the starting of the next cycle is delayed. The amount of delay is unbounded since the batch size is unbounded.

The examples of the cycles with delayed starting are given in Figure 7.21. In the figure, the cycle c_4 is longer than the period p_4 during which the batch for the c_4 is formulated. This causes the starting of c_5 and c_6 to be delayed. As a result, the maximum timing error that can exist during c_5 is *te* in the minimum since the token delivery and the base table access have extra delays.

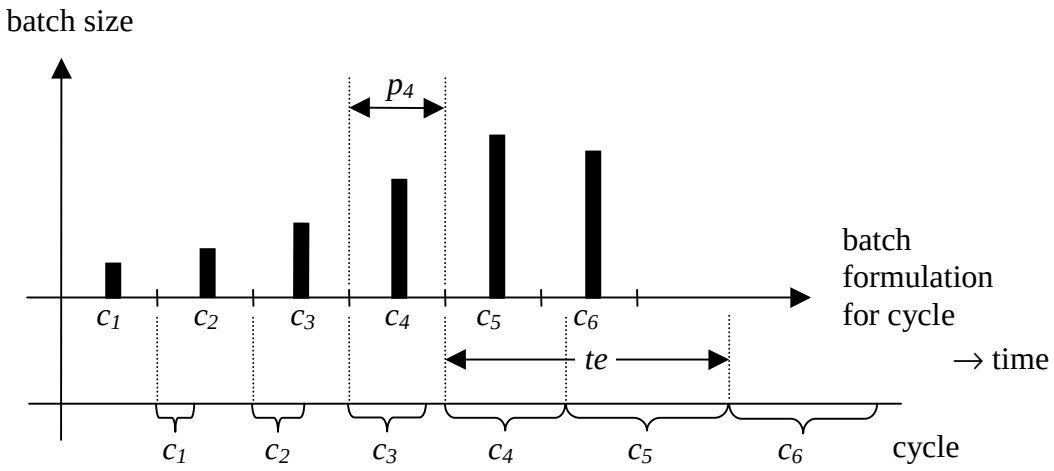


Figure 7.21: Delayed starting of token processing cycles.

When a discrimination network has virtual α nodes, the *timing error* of a cycle is unbounded because of the following facts:

- The delay before the starting of a cycle is unbounded.
- The base table of a virtual α node needs to be accessed when the virtual α node is joined by a token.

The definition of Level 1 consistency requires the *timing error* in the joining tuples to be limited to some fixed values. In fact, we cannot reduce the delay before the starting of a cycle. Therefore, the only way to limit the *timing error* is to remove the necessity of base table access. By employing the shadow table (Subsection 7.3.1), we can remove the necessity of base table access.

In conclusion, to support the virtual α nodes for the triggers with Level 1 consistency, the shadow table needs to be employed. A shadow table is a kind of an α node. Hence, it can be stabilized using Strategy I.

7.5.4 Summary of Level 1 Consistency

Compared to Level 2 consistency, Level 1 consistency allows an untimely joining error to occur. Hence, all the tokens that arrive at the same discrimination network can be processed in parallel. As a result, all the tokens that arrive at the system during a cycle can be processed in parallel, which would increase the performance of the system. We use the reduced computation unit for Level 1 consistency. This reduces the job granularity, which would further increase the overall performance of the system.

However, concurrent token processing can corrupt the content of memory nodes. Therefore, we propose a special mechanism called Stability-Lookaside Buffer (SLB) to make memory nodes stabilize. The concept of an SLB is adopted from the Translation-Lookaside Buffer (TLB) used in the computer hardware arena. To make α nodes stabilize, an SLB is allocated to each α node. We propose Strategy I that includes the content specification for the SLB, the manipulation of the SLB, and the processing of tokens that arrive at the α nodes. We prove the stabilization of α nodes when they are maintained using Strategy I. The proof is given in Theorem 7.1.

To make the β node stabilize, Strategy II, a modified version of Strategy I, is proposed. Strategy II includes the content specification of the SLB for the β nodes, the manipulation of the SLB, the assignment of the timestamps to the components of a compound token, and the token processing policy. We also explained when and how the

dummy timestamp, 0, is used. Lemma 7.1, Lemma 7.2, and Theorem 7.2 prove the stabilization of the β nodes when they are maintained using Strategy II.

Finally, we explained how the delay before the starting of a cycle can be unbounded, and the necessity of shadow tables to support the virtual α nodes of the discrimination network for a trigger with Level 1 consistency.

7.6 Trigger Processing Consistency Level 0

To execute the action of a trigger with Level 0 consistency, the system needs to guarantee that the memory nodes of the discrimination network for the trigger converge. Compared to Level 1 consistency, Level 0 consistency allows the timing error to extend indefinitely. To see how timing error can extend indefinitely, refer to Figure 7.21. With this relaxed consistency requirement, a shadow table is unnecessary for a virtual α node. The reduced computation unit is used for Level 0 consistency. This would increase the performance of the system.

By not employing the shadow tables, another type of *duplicate compound token* problem (Subsection 7.3.2) happens. The problem happens when two joining tokens, tk_1 and tk_2, are from virtual α nodes and one (tk_1) is processed after the other (tk_2) is created and before tk_2 is processed. When tk_2 is processed, it creates the duplicate compound token. Since tk_2 can be processed in a later cycle than the cycle in which tk_1 is processed, the DLB (Subsection 7.3.2) cannot be cleared at the end of cycle. Instead, the DLB needs to be cleared incrementally.

An example of the *duplicate compound token* is given in Figure 7.22 where tk_1 and tk_2 arrive at the stored α node α_1 and the virtual α node α_2, respectively. Assume

both tokens are + tokens and join with each other. If tk_1 is processed after tk_2 is created at the base tables of α_2, and before tk_2 is processed, then a compound token, (+, tk_1, tk_2), will be created when tk_1 is processed. Later, when tk_2 is processed, it will create the same compound token. As a result, two copies of the same compound token will arrive at β_1. This is the *duplicate compound token* problem.

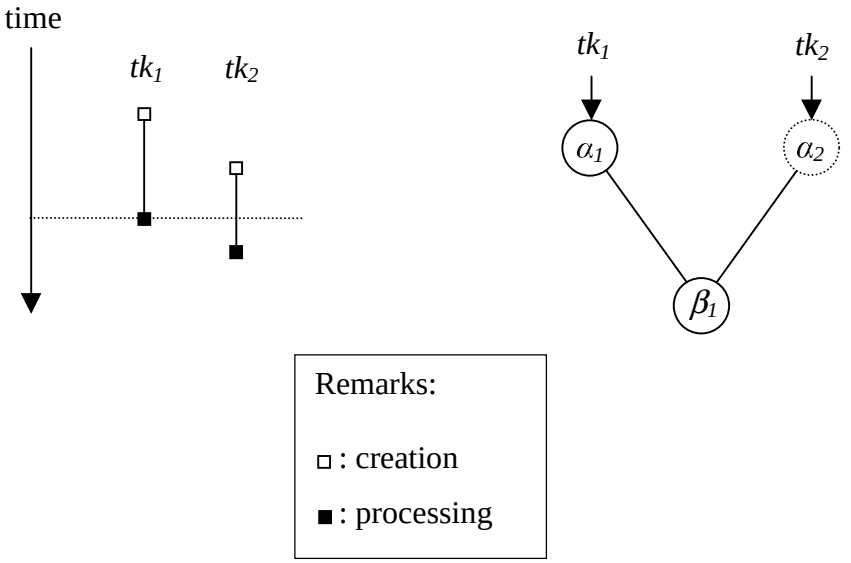


Figure 7.22: Duplicate compound token

Shortly, we will propose Strategy III which makes β nodes that have one or more virtual α node descendents converge. Strategy III includes the content and the type of the tokens that arrive at β nodes, the content and structure of the SLB for β nodes, and the algorithm that applies + and – tokens against β nodes. Under the *quiescence assumption* (Subsection 7.6.1), we prove the stabilization of a β node when the tokens that arrive at the node are processed using Strategy III.

This section is organized as follows: Subsection 7.6.1 defines the terms used in this section (Section 7.6); Subsection 7.6.2 explains the necessity of the dummy timestamp ∞; Subsection 7.6.3 introduces − token processing policy; Subsection 7.6.4 describes Strategy III; Subsection 7.6.5 proves the stabilization of β nodes under *quiescence assumption*; and finally Subsection 7.6.6 summarizes Level 0 consistency.

7.6.1 Definitions

A − atomic token, atk_1, is ***younger than*** a + compound token or a compound SLB line, tk_2, when $ts(atk_1)$ is greater than the timestamp of the matching component of tk_2.

To prove the stabilization of a β node, we assume the data sources of a β node with one or more virtual α node children become quiescent after some time-point. This assumption is called ***quiescence assumption***.

An ***effective timestamp*** is for a compound token component, *c*, from the base table of a virtual α node. An *effective timestamp* is a kind of real timestamp, but it is unknown to the system. If the token *tk* that inserted or modified *c* in the base table is processed in the current or later cycle, then *ets*(*c*) (*effective timestamp* of *c*) is *ts*(*tk*). Otherwise, *ets*(*c*) is 0.

Let c_1 be a component of a compound token, *tk*. Then, c_1 ***contributes to*** a compound tuple, *tp*, when one of the following is true:

- *tk* creates *tp*.
- *tk* replaces *tp* because it is younger than *tp*.
- *tk* is discarded because it is older than *tp*.

- *tk* is discarded because it is incomparable with *tp*, and c_1 is not younger than the matching component of *tp*.

After a component c_1 *contributes to* a tuple, c_1 is not younger than the matching component of the tuple.

7.6.2 Necessity of Dummy Timestamp ∞

Since Strategy III does not employ shadow tables for virtual α nodes, the base table of a virtual α node needs to be accessed when a token joins with the virtual α node. In Strategy II, a dummy timestamp 0 is used for a compound token component that has not been modified in the current cycle. Similarly, a dummy timestamp is needed for the compound token component that is retrieved from the base table of a virtual α node. It is certain that such a component from the base table is not older than any tokens that are delivered to the system, since every token is already applied to the base table. It is possible that the component from a base table is already modified by one or more tokens that are not delivered to the system yet.

The necessity of this new dummy timestamp is illustrated in Figure 7.23. The figure shows the situation after atk_1 is processed against α_1 and then atk_2 is processed against α_2. Atomic + token atk_1 creates tk_1. Atomic δ token atk_2 creates tk_2 and tk_3, because each δ token creates a − token and a + token. We propagate − tokens in atomic form (the reason is explained in Subsection 7.6.3). Tokens tk_1, tk_2, and tk_3 are processed in parallel against β_1. Therefore, an appropriate timestamp needs to be assigned for the tp_1 component of tk_3 so that tk_1 and tk_3 can be compared. Since each delivered token has already been applied to the base table, tp_1 (retrieved from the base table of α_1) in tk_3 must

not be older than tp_1 (a delivered token) in tk_1. Hence, the timestamp of tp_1 in tk_3 must be greater than or equal to tm_1.

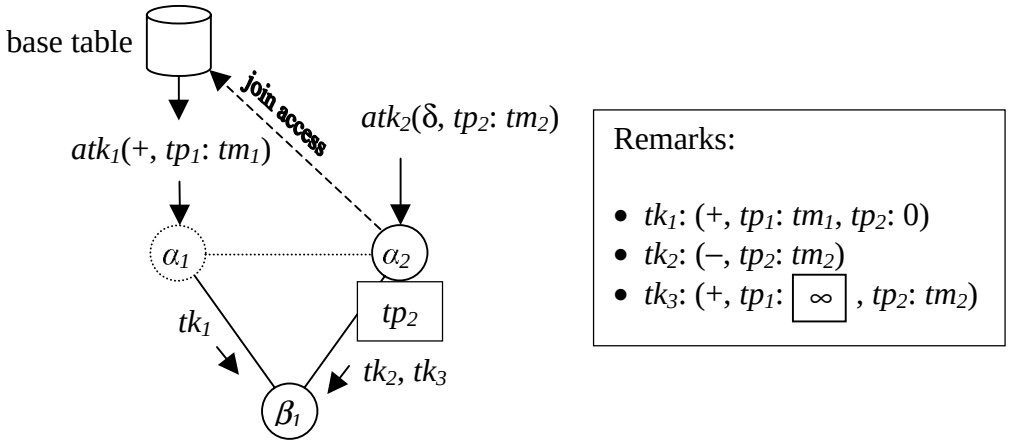


Figure 7.23: The necessity of dummy timestamp ∞

In general, the timestamp of a component (tuple) from a base table must be greater than or equal to the timestamp of any token in the same family. Since the value that is not less than any timestamp is ∞, we will use it as the timestamp for the component (tuple) from a base table of a virtual α node. Note the timestamp of tp_1 in tk_3 is ∞.

7.6.3 Processing of – Tokens

When a – token arrives at a node, *n*, if we join the token with the sibling(s) of *n*, and propagate the results (– compound tokens) to the parent nodes of *n*, then two kinds of anomalies can happen. They are: (1) the creation of a – compound token for a non-

existing tuple and (2) the failure to create a – compound token for the tuple that must be deleted. These anomalies do not happen in Strategy II where shadow tables are employed. The following subsections explain these anomalies and the proposed policy for – token processing.

7.6.3.1 Creation of an unnecessary – compound token

When a tuple, tp_1, in the base table is deleted and a – token, tk_1, for the deletion arrives at a virtual α node, α_1, assume tk_1 joins with the siblings of α_1, and creates a – compound token, tk_3. If tk_3 contains a component that is inserted after tp_1 is deleted in the current cycle, then the target tuple of tk_3 does not exist.

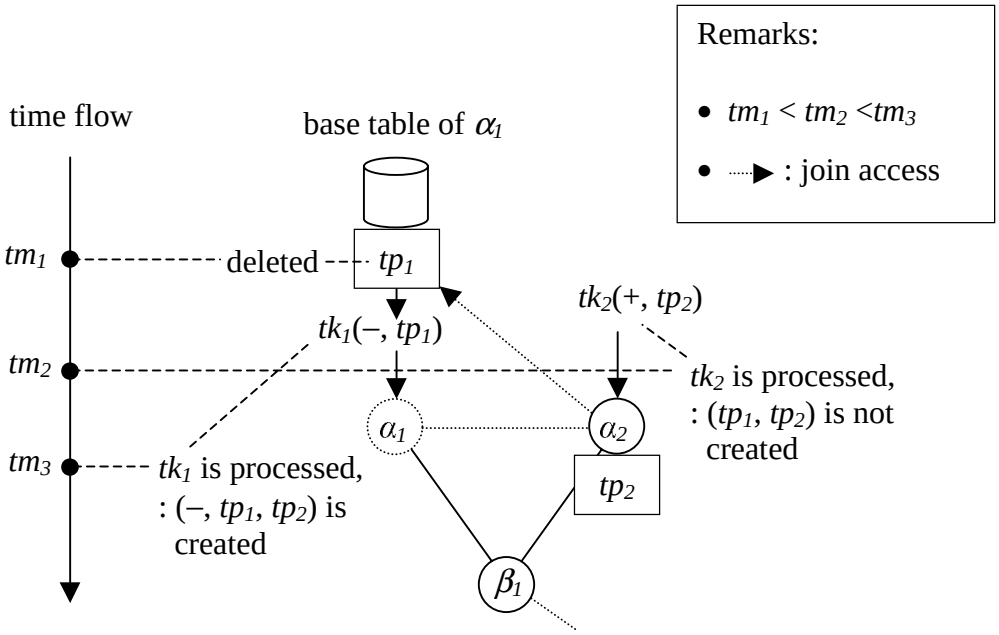


Figure 7.24: Wrong creation of a – compound token.

An example is given in Figure 7.24 where three operations are executed consecutively as follows:

(1) tp_1 is deleted from the base table of a virtual α node α_1, generating tk_1.

(2) tk_2 is processed against α_2, it does not create (tp_1, tp_2).

(3) tk_1 is processed against α_1, it creates a – compound token tk_3(–, tp_1, tp_2).

However, the compound tuple, (tp_1, tp_2), does not exist in β_1 and never will. This is an example of the anomaly of creating a – compound token for a non-existing tuple.

7.6.3.2 Failure to create a necessary – compound token

When a β node, β_1, has two or more virtual α nodes as its children, let tp_3 be a tuple in β_1. Let atp_1 and atp_2 are atomic tuples that constitute tp_3 and are from the virtual α nodes α_1 and α_2, respectively. Let bt_1 and bt_2 are the base tables of α_1 and α_2, respectively. Then, the – compound token whose target is tp_3 will not be created, when atp_1 and atp_2 are deleted satisfying both of the following conditions:

- The token for the deletion of atp_1 from bt_1 is processed against α_1 after atp_2 is deleted from bt_2.
- The token for the deletion of atp_2 from bt_2 is processed against α_2 after atp_1 is deleted from bt_1.

The tuple tp_3 must be deleted, so this is an anomaly of failing to create a necessary – compound token. An example is given in Figure 7.25. When tk_1 is processed against α_1, tp_2 was already been deleted from bt_2. Hence, tk_1 will not create a – compound token that would delete the compound tuple, (tp_1, tp_2), from β_1. The same

case happens when tk_2 is processed against α_2. This is an anomaly of not creating a – compound token for the tuple that must be deleted.

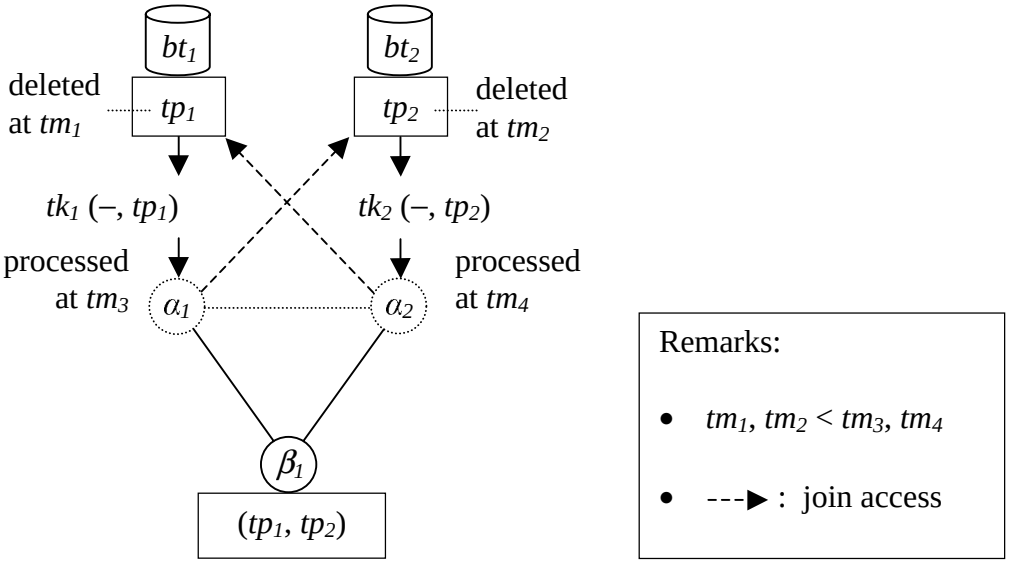


Figure 7.25: Failing to create a necessary – compound token.

7.6.3.3 Policy of – token processing

By propagating – atomic tokens, the two anomalies will be removed as follows:

- Since a – token will not be joined with the base tables of the virtual α nodes, the – compound token that deletes a non-existing tuple will not be created.
- Since the node *n* at which a – atomic token, *atk*, arrives will be searched to find the compound tuple that contains *atk*, each and every compound tuple that must be deleted will be. However, to prevent compound tuples that are younger than *atk* from being deleted, the compound SLB of *n* needs to be

checked. This checking is done by the line (8) of *minus_token_processing* in Figure 7.27.

Therefore, in Strategy III, where shadow tables are not employed for the virtual α nodes, when a – atomic token arrives at a memory node *n*, the – atomic token itself will be propagated to the parent nodes of *n*. By dong this two kinds of anomalies can be prevented.

7.6.4 Strategy III

Included in Strategy III for the stabilization of β nodes are:

(1) The assignment of timestamps to the + compound tokens.

(2) The SLB structure for β nodes.

(3) The algorithms that apply tokens against β nodes.

These will be explained in turn in the following subsections.

7.6.4.1 Timestamp assignment to a + compound token

Let tk_1 be a token that arrives at a node, *n*, and let tk_2 be a token that is created as a propagation of tk_1 to a parent node of *n*. Then the component *c* of tk_2 will be assigned timestamps using the as following policies:

- If *c* is from a stored α node and a line, *l*, that is in the same family as *c* exists in the SLB of the α node, then the timestamp of *l* is assigned to *c*.
- If *c* is from a stored α node and a line that is in the same family as *c* does not exist in the SLB of the α node, then 0 is assigned to *c*.
- If *c* is from the base table of a virtual α node, then ∞ is assigned to *c*.

7.6.4.2 SLB structure for a β node

In Strategy III, each β node is equipped with a compound SLB and one atomic SLB per α node that is a descendent of the β node. The compound SLB is for the + compound tokens while the atomic SLB is for the – atomic tokens. Each line in a compound SLB contains the key and the timestamp of each atomic component of the youngest + compound token in a family that has inserted a tuple in the current cycle. Each line in an atomic SLB contains the key and the timestamp of the youngest – atomic token in a family that has been seen in the current cycle.

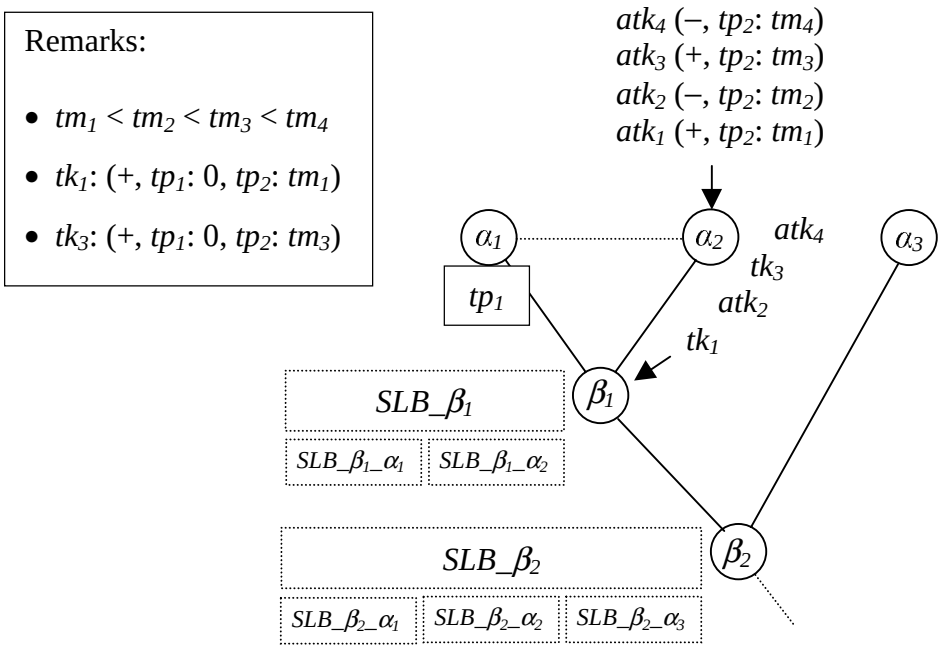


Figure 7.26: The SLB structure for β nodes.

An example is given in Figure 7.26 where the situation after four tokens, atk_1, atk_2, atk_3, and atk_4, are consecutively processed against α_2 and create the propagation of tk_1, atk_2, tk_3, and atk_4, respectively, as shown. The processing of the four tokens that arrive at β_1 is summarized in Table 7.6.

Table 7.6: Processing of four tokens that arrive at β_1.

processing order	token	effect of token processing	affected SLB
1	tk_1	create a line l_1 containing "tp_1: 0, tp_2: tm_1"	SLB_β_1
2	atk_2	create a line l_2 containing "tp_2: tm_2"	$SLB\_\beta_1\_\alpha_2$
3	tk_3	modify l_1 into "tp_1: 0, tp_2: tm_3"	SLB_β_1
4	atk_4	modify l_2 into "tp_2: tm_4"	$SLB\_\beta_1\_\alpha_2$

7.6.4.3 Token processing algorithm

There are two types of tokens that arrive at a β node. Hence, we provide two procedures *minus_token_processing* and *plus_token_processing*. The procedure *minus_token_processing* that processes – tokens is given in Figure 7.27. The detailed reasoning for the key steps of the procedure is given in the following paragraph.

Since, in Strategy III, – atomic tokens are propagated as atomic tokens, a – token can affect multiple β node tuples. Furthermore, when a – token arrives at a β node, some

tuples are younger than the token while others are older. Therefore, to protect the younger tuples from being deleted by an older – token, lines from (8) to (10) are needed in *minus_token_processing*.

```
procedure minus_token_processing (token t, node β1)
// t: a – token (assume t originally arrived at an α node α1)
// SLB_β1: the compound SLB of β1
// SLB_β1_α1: an atomic SLB of β1 for the – tokens propagated from α1
 (1)  if SLB_β1_α1 has a line, l, that is in the same family as t
 (2)      then if ts(l) < ts(t) // t is younger than l
 (3)               then replace the timestamp of l with ts(t)
 (4)               else discard t and end this procedure // l is younger than t
 (5)      else initialize a line for t in SLB_β1_α1
 (6)  for each compound tuple c of β1 such that c has t as its component
 (7)      if c has a line, l2, in SLB_β1
 (8)          then if ts(l2.t) < ts(t)
 (9)                    then delete c
(10)                    else do nothing // c is younger that t
(11)          else delete c
(12)  propagate t to each parent node of β1
end minus_token_processing
```

Figure 7.27: Procedure *minus_token_processing*.

An example is given in Figure 7.28 (a) where four tokens, atk_1, atk_2, atk_3, and atk_4, arrive at α_1 and α_2. Before the arrival of the four tokens, β_1 contains two compound

tuples (tp_1, tp_2) and (tp_1, tp_3). If the four tokens are processed in the increasing order of their subscripts, then the tokens that arrive at β_1 are listed in Table 7.7.

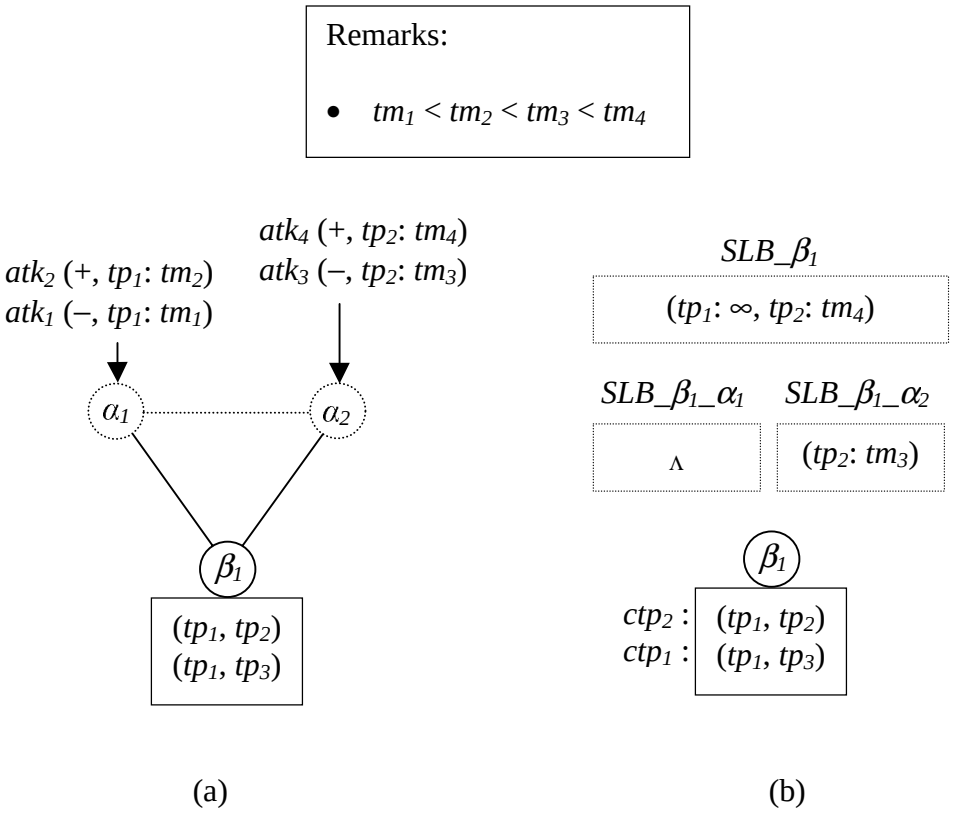


Figure 7.28: A compound tuple that is younger than a – token.

While the tokens in Table 7.7 are processed in parallel, assume that the tokens are processed in the order of atk_3, tk_4, atk_1, etc. Then, right after tk_4 is processed the SLBs and the content of β_1 is shown in Figure 7.28 (b). Because the compound tuple ctp_2 in β_1 is younger than atk_1 (see SLB_β_1 in Figure 7.28), tp_2 should not be deleted. However, tp_1 is older than atk_1, and needs to be deleted. This is the situation where the lines from 8 to 10 of *minus_ token_processing* are necessary.

Table 7.7: Tokens that arrive at β_1

Arrival order	Processing order	Token	Propagated from
4	2	tk_4 (+, tp_1: ∞, tp_2: tm_4)	atk_4
3	1	atk_3 (–, tp_2: tm_3)	atk_3
2	.	tk_{22} (+, tp_1: tm_2, tp_3: ∞)	atk_2
	.	tk_{21} (+, tp_1: tm_2, tp_2: ∞)	
1	3	atk_1 (–, tp_1: tm_1)	atk_1

The procedure *plus_token_processing* that processes + tokens is given in Figure 7.29. Lines 10 and 11 are needed in *plus_token_processing* since the tokens are processed in parallel. This is because the + compound token that inserts a tuple could be executed before the – atomic token that deletes the previous version of the tuple has done so.

7.6.5 Proof of β Node Stabilization in Strategy III

When a β node does not have a virtual α node as its child, its content at the end of each cycle is determined solely based on the tokens that arrived at the β node from its children. However, when a β node has one or more virtual α nodes as its children, its content at the end of each cycle is also determined by the changes that happen upon the base tables of the virtual α nodes.

```
Procedure plus_token_processing (token t, β_node β1)
// t: a + token
// SLB_β1: the compound SLB of β1
// SLB_β1_αi: an atomic SLB of β1 for the – tokens propagated from αi
 (1)  for each SLB_β1_αi of β1
 (2)      if (SLB_β1_αi has a line l for t.i) // t.i is the i-th component of t
 (3)          then if ts(l) > ts(t.i) // l is younger than t
 (4)                   then discard t and return from this procedure
 (5)  if SLB has a line, l, that is in the same family as t
 (6)      then if ts(l) < ts(t.i) // t is younger than l
 (7)               then update l using t
 (8)               else discard t and return from this procedure
 (9)      else create a line using t in SLB_β1
(10)  if (a tuple, tp2, with the same key as t exists in β1)
(11)      then create a – token for tp2 and process and propagate it
(12)  insert t into β1
(13)  propagate t to the parent nodes of β1
end plus_token_processing
```

Figure 7.29: Procedure *plus_token_processing*

An example is given in Figure 7.30 where the situation when the following two operations are about to execute is depicted:

- The application of tk_1 against α_2.
- The modification of tp_1 into tp_1' at the base table.

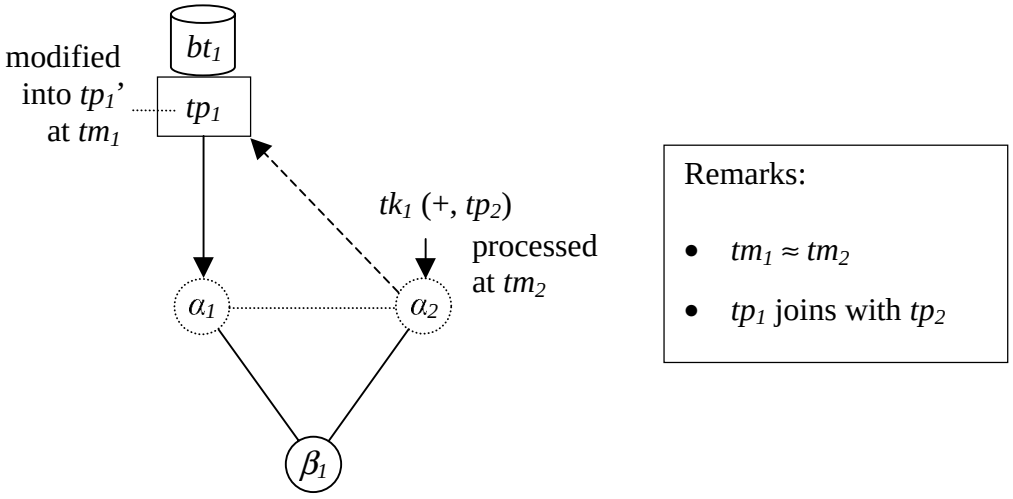


Figure 7.30: Stabilization criteria of a β node

After tk_1 is processed, what compound tuple will be inserted into β_1? Will it be (tp_1, tp_2) or (tp_1', tp_2)? It simply depends on the relative execution order of the above two operations. The content of the β node that has one or more virtual α nodes as its children depends on the modification of the base tables of the virtual α nodes. In short, there are no stabilization criteria existing for the β nodes that have one or more virtual α nodes among their children. Therefore, we need the *quiescence assumption* (see Subsection 7.6.1) to create the stabilization criteria and prove that Strategy III makes a β node that has one or more virtual α node descendents stabilize.

Under the *quiescence assumption*, the proof of the β node stabilization is embodied in the following lemmas and Theorem 7.3.

Lemma 7.3: When a – atomic token, tk_1, arrives at a β node, β_1, if tk_1 is processed using Strategy III (procedure *minus_token_processing*), then tk_1 deletes each β_1 tuple that is older than tk_1.

Proof: Assume α_1 is a descendent of β_1 and is the α node at which tk_1 initially arrived. Let $SLB\_\beta_1\_\alpha_1$ belong to β_1 to store the – tokens from α_1. If $SLB\_\beta_1\_\alpha_1$ does not have a line that belongs to the same family as tk_1, or $SLB\_\beta_1\_\alpha_1$ has a line that belongs to the same family as tk_1 and is older than tk_1, then tk_1 will be registered in $SLB\_\beta_1\_\alpha_1$ and *minus_token_processing* will locate each β_1 tuple, *tp*, that has tk_1 as a component. Let SLB_β_1 be the compound SLB of β_1. Then, *tp* can occur in one of three cases:

(i) No line in SLB_β_1 is in the same family as *tp*.

(ii) A line in SLB_β_1 is in the same family as *tp* and is younger than tk_1.

(iii) A line in SLB_β_1 is in the same family as *tp* and is older than tk_1.

In case (i), *tp* should be deleted, because it must have been inserted in some previous cycle, which means *tp* is older than tk_1. In case (ii), *tp* needs to be left untouched since it is younger than tk_1. In case (iii), *tp* should be deleted. The cases of (i), (ii) and (iii) are implemented by *minus_token_processing* in lines (11), (10), and (9), respectively. Therefore, when tk_1 is registered in $SLB\_\beta_1\_\alpha_1$, tk_1 is proven to delete β_1 tuples that are older than tk_1.

If $SLB\_\beta_1\_\alpha_1$ has a line, *l*, that belongs to the same family as tk_1 and is younger than tk_1, then the tuples that are older than tk_1 must have been deleted when *l* was

registered. This is verified by the discussion in the preceding paragraphs. As a result, tk_1 can be safely discarded. Line (4) of *minus_token_processing* does this. □

Lemma 7.4: Let tk_1 and tk_2 be + compound tokens that are created using Strategy III and arrive at a β node that has one or more virtual α nodes as its children. If tk_1 and tk_2 belong to the same family and are *incomparable*, then the four components of them (two from tk_1 and two from tk_2) that make tk_1 and tk_2 incomparable will have the values and relative magnitudes that are shown in Figure 7.31 (a) or (b). The following acronyms are used in the figure:

- cu_1 and cu_2: The computation units that created tk_1 and tk_2, respectively.
- c_{11} and c_{12}: The components of tk_1 that make tk_1 and tk_2 incomparable.
- c_{21} and c_{22}: The components of tk_2 in the same position as c_{11} and c_{12}, respectively.
- α_1: The α node to which c_{11} and c_{21} are from ($ts(c_{11}) > ts(c_{21})$).
- α_2: the α node to which c_{12} and c_{22} are from ($ts(c_{12}) > ts(c_{22})$).

Proof: In terms of node type, α_1 and α_2 belong to one of three cases:

Case I: Both α_1 and α_2 are stored α nodes.

Case II: One of α_1 and α_2 is a stored α node and the other is a virtual α node.

Case III: Both α_1 and α_2 are virtual α nodes.

Case I: If cu_1 was executed first, then later when cu_2 is executed, the system should not see c_{21} that is older than c_{11}. Therefore, tk_2 cannot be created. Similarly, when cu_2 was executed first, tk_1 can not be created. In short, tk_1 and tk_2 cannot coexist. As a result, both α_1 and α_2 cannot be stored α nodes at the same time and Case I does not exist.

	α_1^v	α_2
tk_1/cu_1	$ts(c_{11}) = \infty$	$ts(c_{12})$
	$>$	$<$
tk_2/cu_2	$ts(c_{21}^i)$	$ts(c_{22}) \neq \infty$

(a) Case II

	α_1^v	α_2^v
tk_1/cu_1	$ts(c_{11}) = \infty$	$ts(c_{12}^i)$
	$>$	$<$
tk_2/cu_2	$ts(c_{21}^i)$	$ts(c_{22}) = \infty$

(b) Case III

^{v} virtual α node
^{i} initiating token

Figure 7.31: Four components of two incomparable tokens.

Case II: We can say that α_1 is a virtual α node without loss of generality. Then c_{21} must be the *initiating token* of tk_2 since it is from α_1 and $ts(c_{21})$ is non-∞.

Assume that $ts(c_{11})$ is non-∞. Then c_{11} must be the *initiating token* of tk_1. If cu_1 was executed first, then later when c_{21} is processed, the system discards c_{21} since it is older than c_{11}. Therefore, tk_2 cannot be created. If cu_2 was executed first, then when cu_1 is executed, the system should not see c_{12} that is older than c_{22}. Therefore, tk_1 cannot be created. In short, tk_1 and tk_2 cannot coexist under the assumption of non-∞ $ts(c_{11})$. It is a contradiction. Therefore, $ts(c_{11})$ is ∞. This case is shown in Figure 7.31 (a), where $ts(c_{22})$ is non-∞ since α_2 is a stored α node.

Case III: Here, $ts(c_{12})$ is *non*-∞ since it is less than at least one value: $ts(c_{22})$. Therefore, c_{12} is the initiating token of tk_1 because α_2 is a virtual α node. Since c_{11} is from a virtual

α node and is not the initiating token, $ts(c_{11})$ is ∞. Similarly, $ts(c_{21})$ is non-∞ and $ts(c_{22})$ is ∞. This case is shown in Figure 7.31 (b). This concludes the proof of Lemma 7.4. □

Corollary Let tk_1 and tk_2 be + compound tokens in the same *family*. Let c_{11} and c_{12} be components of tk_1. Let c_{21} and c_{22} be components of tk_2 corresponding to c_{11} and c_{12}, respectively. Given that $ts(c_{11})$, $ts(c_{12})$, $ts(c_{21})$, and $ts(c_{22})$ are non-∞, then $ts(c_{11}) > ts(c_{21})$ implies $ts(c_{12}) \geq ts(c_{22})$.

Lemma 7.5: Let tk_1 and tk_2 be + compound tokens in the same family. Let c_1 and c_2 be components in the same position of tk_1 and tk_2, respectively. Then, tk_1 is younger than tk_2 if c_1 and c_2 satisfy the following two conditions:

- c_1 is the initiating token of tk_1.
- $ts(c_1)$ is greater than $ts(c_2)$.

Proof: By the Corollary of Lemma 7.4 and the fact that $ts(c_1)$ is greater than $ts(c_2)$, the timestamp of any tk_1 component (other than c_1) that is from a stored α node, is greater than or equal to the timestamp of the corresponding component of tk_2. The timestamp of any tk_1 component (other than c_1) that is from a virtual α node is ∞ since c_1 is the initiating token of tk_1, and $ts(c_1)$ is greater than $ts(c_2)$ by the condition of Lemma 7.5.

Therefore, the timestamp of each tk_1 component is greater than or equal to the timestamp of the corresponding component of tk_2. Hence, tk_1 is younger than tk_2. □

Lemma 7.6: Let tk_1 be a + compound token that arrives at a β node, β_1. Assume the compound SLB of β_1 has a line, *l*, that is in the same family as tk_1. If tk_1 is discarded since it is incomparable with *l*, then each tk_1 component that is younger than the corresponding *l* component eventually *contributes to* (Subsection 7.6.1) tp_1.

Proof: As the proof, first we will show that for each tk_1 component c_1 which is younger than the corresponding l component lc_1, a + compound token tk_2 that contains c_1 exists and is not processed yet. Next, we will show that c_1 *contributes to* tp_1 when tk_2 is processed.

The timestamp pair that c_1 and lc_1 will have can fall under one of the four cases in Figure 7.32. For each case, when c_1 is younger that lc_1 (when row *condition* is true in the figure), we will prove that tk_2 exists and is not processed yet.

Case	I	II	III	IV
$ts(c_1)$	$\neq \infty$	∞	$\neq \infty$	∞
$ts(lc_1)$	$\neq \infty$	$\neq \infty$	∞	∞
Condition	$ts(c_1) > ts(lc_1)$	$ets(c_1) > ts(lc_1)$	impossible	impossible

Figure 7.32: Four cases to which a timestamp pair can belong.

Case I. For tk_1 to be incomparable with l, another component pair, c_2 and lc_2, should exist and satisfy $ts(c_2) < ts(lc_2)$ where c_2 and lc_2 are components of tk_1 and l, respectively. Two possible combinations of timestamps of c_2 and lc_2 are: (1) $ts(c_2) \neq \infty$ and $ts(lc_2) \neq \infty$, and (2) $ts(c_2) \neq \infty$ and $ts(lc_2) = \infty$. The combination (1) was proven to be impossible by Lemma 7.4 and Case I. Therefore, the initiating token of tk_1 is not c_1 but c_2, because c_2 is from a virtual α node and $ts(c_2)$ is non-∞. There must be a compound token, tk_2, that has c_1 as its initiating token. Since c_1 is the initiating token of tk_2 and c_1 is younger than lc_1, tk_2 is younger than l by Lemma 7.5. If tk_2 is already processed against β_1, l must have

been updated by tk_2, because tk_2 is younger than l. Hence, tk_2 exists and is not processed yet.

Case II. Since $ts(c_1)$ is ∞, c_1 is not the initiating token of tk_1. Therefore, a compound token tk_2 that has c_1 as its initiating token must exist. Token tk_2 could be processed in either the current or the next cycle. Assume tk_2 belongs to the current cycle. Since c_1 is the initiating token of tk_2, and c_1 is younger than lc_1, tk_2 is younger than l by Lemma 7.5. If tk_2 is already processed against β_1, l must have been updated by tk_2, because tk_2 is younger than l. Hence, tk_2 exists and is not processed yet. If tk_2 is to be processed in the next cycle, then, obviously, tk_2 will exist and is not processed yet.

Case III & IV. In these cases, c_1 cannot be younger than lc_1. Therefore, we do not need to consider them.

Now, we will show that c_1 will *contribute to* tp_1 when tk_2 is processed later. The token tk_2 can be processed in three ways: it can be applied to tp_1, be discarded since it is older than tp_1, or be discarded since it is incomparable with tp_1. If tk_2 is applied to tp_1 or discarded because it is older than tp_1, then c_1 *contributes to* tp_1 by the definition of *contribute to.*

Let lc_1 be a tp_1 component that corresponds to c_1 of tk_2. By Lemma 7.5 and the fact that c_1 is the initiating token of tk_2, if $ts(c_1) > ts(lc_1)$, then tk_2 is younger that tp_1. Therefore, if tk_2 and tp_1 are incomparable, that means $ts(c_1) \leq ts(lc_1)$. Hence, if tk_2 is discarded since it is incomparable with tp_1, then c_1 is not younger than lc_1 and c_1 *contributes to* tp_1. □

To explain Lemma 7.7, we need to define the term *extended atomic token family* (*extended family*). Let β_1 be a β node in a discrimination network N_1. The *extended family* of the *i*-th component c_i of a tuple, tp_1, in β_1 is a set of atomic tokens that have the following properties:

- They are generated at the data source ds_i for c_i (the data source of the *i*-th α node descendent of β_1) from right after N_1 is created until ds_i becomes quiescent (see Figure 7.30 for more details).
- In addition, they are either:
  - a – token that has the same key as c_i and arrives at β_1 or
  - a + token that is embedded in a compound token that arrives at β_1 and has the same compound key as tp_1.

For example, the extended families for the tuples in a β node, β_1 (Figure 7.33), include the tokens that are created at the data sources that feed the α node descendents of β_1 during all *k* cycles. In the figure, we assume the data sources become quiescent after the time *tm*.

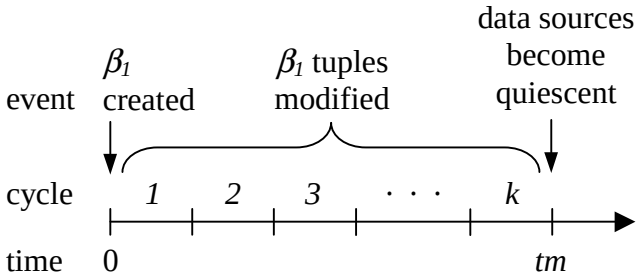


Figure 7.33: The creation and modification cycle of β node β_1.

Each component of a β node tuple is from a data source and has a key. Therefore, an extended family is associated with each component of a β node tuple. An example is given in Figure 7.34 where β_1 has n α node descendents. Then, *n extended families* are associated with each β_1 tuple. In the figure, the extended family $efam_i$ is associated with the component c_i of the tuple tp_1. The $efam_i$ contains atomic tokens that have the same key as c_i. The tokens were made in the data source of α_i, ds_i, after β_1 was created and before ds_i became quiescent.

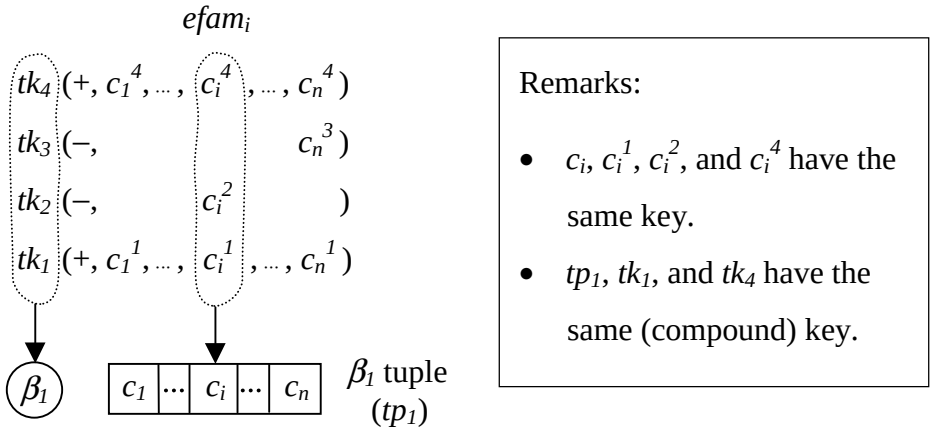


Figure 7.34: The association of extended families with β tuple components.

Lemma 7.7: Let tp_1 be a compound tuple of a β node, β_1, and let tk_1 be a + compound token that arrives at β_1, and let tk_1 be in the same family as tp_1. If tk_1 is younger than each − token that is in any extended family associated with tp_1, then each component of *tk_1 contributes to tp_1* after tk_1 is processed against β_1 using Strategy III.

Proof: Since tk_1 is younger than any − token, line 4 of *plus_token_processing* (Figure 7.29) will be skipped when tk_1 is processed. If no SLB line that is in the same family as

tp_1 exists, then tk_1 will be inserted into β_1 by line 12 of *plus_token_processing*. By the definition of *contribute to*, each component of tk_1 *contributes to* tp_1.

If an SLB line, *l*, is in the same family as tp_1, depending on the timestamps of tk_1 and *l*, the action of *plus_token_processing* differs as follows:

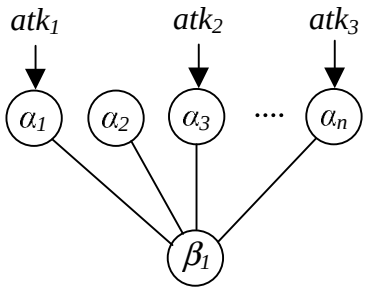
- When tk_1 is younger than *l*, tk_1 is inserted into β_1 by line 12 of *plus_token_processing*. Therefore, by the definition of *contribute to*, each component of tk_1 *contributes to* tp_1.
- When tk_1 is older than *l*, tk_1 is discarded by line 8 of *plus_token_processing*. Therefore, by the definition of *contribute to*, each component of tk_1 *contributes to* tp_1.
- When tk_1 is incomparable with *l*, tk_1 is discarded by line 8 of *plus_token_processing*. By the definition of *contribute to*, any tk_1 component that is not younger than the matching component of tp_1 *contributes to* tp_1. If the component *c* of tk_1 is younger than the corresponding components of tp_1, then *c* will contribute to tp_1. This is proven by Lemma 7.6. Therefore, each component of tk_1 contributes to tp_1.

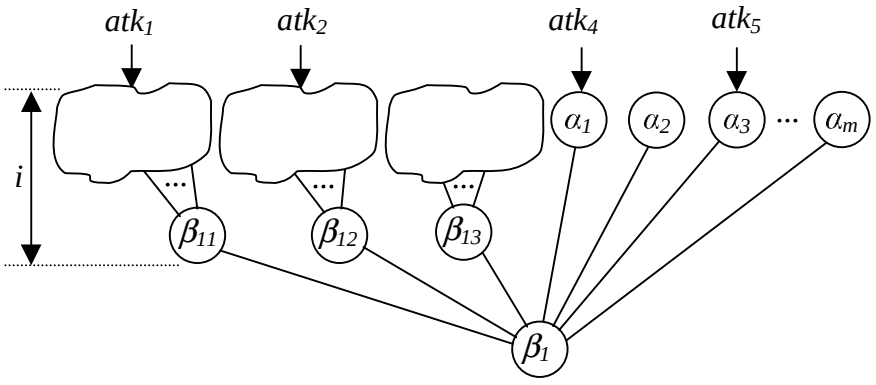
Whether or not an SLB line exists for tp_1, we have proven that each component of tk_1 contributes to tp_1. Therefore, Lemma 7.7 is proven. □

Lemma 7.8: Let β_1 be a β node and tp_1 be a tuple of β_1. If at least one extended family that is related with tp_1 is non-empty and the youngest tokens of each non-empty extended family are + atomic tokens, then a + *compound token that contains all the youngest tokens* (the ***youngest + compound token***) will arrive at β_1.

Proof: The height of β_1 in a Gator network is the length of a longest path from β_1 to an α node. Using mathematical induction on the height (h) of β_1, we will prove that the *youngest + compound token* (*yctk*) will arrive at β_1.

(a)





(b)

Figure 7.35: The creation of *the youngest + compound token*

Basis: When h is 1 (See (a) of Figure 7.35), let atk_1 be the last processed token among the youngest tokens that arrive at the children of β_1. Without loss of generality we can say that atk_1 arrives at α_1. When atk_1 is processed against α_1, the other youngest tokens (for example, atk_2 and atk_3 in Figure 7.35 (a)) that arrive at other children of β_1 must have

been processed against the related nodes (for example, α_3 and α_n) since atk_1 is processed later than them. Therefore, as a propagation of atk_1, *yctk* will be created and arrive at β_1.

Induction: Suppose that the hypothesis is true when *h* is *i*. Let the height of β_1 be *i*+1. Then the heights of the children of β_1 *are i* or less (see Figure 7.35 (b)). By the induction assumption, *the yctk*s arrive at all the β node children of β_1. Let tk_1 be the last processed token among (i) the youngest tokens that arrive at α node children of β_1 (for example, α_1 and α_3 in Figure 7.35 (b)) and (ii) *the yctk*s that arrive at the β node children of β_1 (β_{11} and β_{12} in Figure 7.35 (b)). When tk_1 is processed against the related child node of β_1, the youngest tokens and/or the *yctk*s that arrive at other children of β_1 must have been processed against the related nodes since tk_1 is processed later than them. Therefore, as a propagation of tk_1, *yctk* will be created and arrive at β_1. Therefore, Lemma 7.8 is proven. □

Theorem 7.3: Let β_1 be a β node that has one or more virtual α nodes among its descendents. Under the *quiescence assumption*, if the tokens that arrive at β_1 are processed using Strategy III, then β_1 stabilizes after all the tokens that arrive at β_1 are processed.

Proof: The β node β_1 stabilizes if and only if each tuple in it stabilizes after all the tokens from the data sources that feed β_1 are processed. If β_1 has *n* α node descendents, then *n* extended families are associated with each β_1 tuple tp_1. Some of the extended families could be empty. To prove the theorem, we need to establish that the youngest tokens of the non-empty extended families *contribute to* tp_1 by proving the following:

(i) If one or more youngest tokens are – tokens, then tp_1 is deleted.

(ii) If all youngest tokens are + tokens, then tp_1 exists and contains all the youngest tokens as its components.

When one or more youngest tokens are – tokens, let tk_1 be the first processed token among the – tokens. Since tk_1 is the youngest token, tk_1 is younger than tp_1. By Lemma 7.3 and the fact that tk_1 is younger than tp_1, tk_1 deletes tp_1. Therefore, (i) is proven.

By Lemma 7.8, when all youngest tokens are + tokens, a + compound token, tk_2, that contains all the youngest tokens will arrive at β_1. By Lemma 7.7, it is proven that each component of tk_2 *contributes to* tp_1. Note that tp_1 cannot contain components that are younger than the corresponding components of tk_2, since tk_2 contains all the youngest tokens. Therefore, (ii) is proven. Now it is established that the youngest tokens of the non-empty extended families *contribute to* tp_1. Hence, the theorem is proven. □

7.6.6 Summary of Level 0 Consistency

Compared to Level 1 consistency, Level 0 consistency removes the necessity of shadow tables by allowing the *timing error* to grow indefinitely. However, Level 0 consistency requires the memory nodes of a discrimination network converge just as in Level 1 consistency. Therefore, to stabilize the β nodes that have one or more virtual α node descendents, we propose Strategy III. The key features of the Strategy III are:

- The dummy timestamp ∞ is used for the compound token components that are retrieved from the base table of a virtual α node.
- To remove anomalies, – tokens are propagated as atomic tokens.

- Each β node is equipped with two kinds of SLBs: a compound SLB to store + compound tokens and an atomic SLB for each α node descendent of the β node to store – atomic tokens.
- Two procedures, *minus_token_processing* for the – token processing and *plus_token_processing* for the + token processing, are proposed.

Finally, under the *quiescence assumption* of the data sources that feed a β node, we prove the stabilization of a β node. The proof is embodied in six lemmas and Theorem 7.3.

7.7 Implementation Alternatives

The type of discrimination network influences the performance and the complexity of system significantly. Therefore, to help the system developers, we examine four kinds of discrimination networks:

- Pure virtual TREAT network
- TREAT network
- A–TREAT network
- Gator network

For each of the above discrimination network, we will discuss the achievable levels of consistency and the way to achieve them. For the sake of simplicity, *serial token processing* means *serial processing of the tokens that arrive at the same discrimination network.*

7.7.1 Pure Virtual TREAT Network

By pure virtual TREAT, we mean the TREAT (Section 2.1) with virtual α nodes only. Therefore, in pure virtual TREAT, a shadow table (Subsection 7.3.1) cannot be employed, since a shadow is a kind of stored table. If we use pure virtual TREAT, the *untimely joining error* is unavoidable, and the *timing error* is unlimited. Hence, the highest consistency level that can be achieved with this alternative is Level 0.

Even when the tokens are processed in parallel, the SLB mechanism (sections 7.5 and 7.6) is unnecessary since there are no memory nodes to maintain. The chance of untimely joining error could be increased due to the parallel token processing. However, the amount of timing error could be decreased as a result of parallel token processing because it will often take less time to process an arriving token. The biggest benefit of parallel token processing is the increased overall performance of the system.

The advantages and disadvantages of pure virtual TREAT compared to other discrimination networks are as follows:

- Advantages:
  - No storage to store the memory nodes is needed.
  - Tokens can be processed in parallel without fear of corrupting stored memory nodes, because they do not exist. Hence, we do not need to resort to the complex SLB mechanism.
- Disadvantages:
  - Only Level 0 consistency can be provided.

7.7.2 TREAT Network

In a TREAT network each α node is a stored α node. Therefore, the timing error in the joining tuples is limited to the duration during which a batch is formulated. By using TREAT network as a discrimination network, we can achieve three trigger processing consistency levels as follows:

- Level 3: Process tokens serially.
- Level 2: Use CTS detection (Subsection 7.4.1), parallel token processing within a CTS.
- Level 1: Perform parallel token processing using Strategy I (Subsection 7.5.1.1) for α nodes.

The advantages and disadvantages of TREAT compared to pure virtual TREAT network are as follows:

- Advantages:
  - No duplicate compound token problem will occur.
  - Consistency level 3, 2, and 1 can be provided.
- Disadvantages:
  - Storage to store all the memory nodes will be huge.

7.7.3 A–TREAT Network

A–TREAT [33] is a modification of TREAT (Section 2.1) by adding the virtual α nodes. By using an A–TREAT network as a discrimination network, we can achieve each trigger processing consistency level as follows:

- Level 3: Process tokens serially, and use shadow tables for virtual α nodes.
- Level 2: Use CTS detection, parallel token processing within a CTS, and shadow tables for virtual α nodes.
- Level 1: Perform parallel token processing using Strategy I (Subsection 7.5.1.1) for α nodes, and use shadow tables for virtual α nodes.
- Level 0: Perform parallel token processing using Strategy I (Subection 7.5.2) for α nodes.

The advantages and disadvantages of A−TREAT, compared to TREAT network, are as follows:

- Advantages:
  - The storage requirement will be reduced with shadow tables.
  - Level 3, Level 2, Level 1, and Level 0 consistencies can be provided.
- Disadvantages:
  - Due to shadow tables, system complexity will increase.

7.7.4 Gator Network

Gator (Section 2.1) is a kind of a discrimination network with α nodes and β nodes. An α node in a Gator network can be either a stored α node or a virtual α node. By using Gator network as a discrimination network, we can achieve each trigger processing consistency level as follows:

- Level 3: Process tokens serially, and use shadow tables for virtual α nodes.

- Level 2: Use CTS detection, parallel token processing within a CTS, and shadow tables for virtual α nodes.
- Level 1: Perform parallel token processing using Strategy I for α nodes, perform parallel token processing using Strategy II (Subsection 7.5.2.2) for β nodes, and use shadow tables for virtual α nodes.
- Level 0: Perform parallel token processing using Strategy I (Subsection 7.5.1.1) for α nodes, perform parallel token processing using Strategy II for β nodes that does not have virtual α node descendents, and perform parallel token processing using Strategy III (Subsection 7.6.4) for β nodes that have virtual α node descendents.

The advantages and disadvantages of Gator compared to other discrimination networks are as follows:

- Advantages:
  - Gator network join structures can be optimized [9]. This will improve the overall performance of the system.
- Disadvantages:
  - The system complexity will be the highest.

The discrimination networks listed above have different complexities. From a different standpoint, they are the development steps of a trigger system. We recommend to implement the simplest approach (pure virtual TREAT network) first, and add complexity until the most efficient approach (Gator) is implemented.

7.8 Summary

To exploit the capacity of parallel computers and improve the overall performance of the system, parallel token processing is essential. However, parallel token processing causes semantic problems in trigger processing. Nevertheless, performance improvement through the parallel token processing is worthwhile and some semantic problems caused by parallel token processing could be allowed for some trigger users. Therefore, we defined four consistency levels of trigger processing so that the users can choose the appropriate levels of their triggers. The higher the level is, the more exact the trigger processing semantics are, and the lower the performance is.

Level 3 consistency provides the highest semantic consistency in trigger processing. The tokens that arrive at different discrimination networks can be processed in parallel. However, Level 3 consistency would produce the lowest performance since the tokens that arrive at the same discrimination network need to be processed in serial. We developed a special technique called a *shadow table* to support virtual α nodes and guarantee Level 1 consistency. The sharing of a shadow table among multiple virtual α nodes causes the duplicate compound token problem. To remove the problem, we developed a DLB which a variation of an SLB.

The requirement of serial trigger action execution is removed in Level 2 consistency. However, an *untimely joining error* is not allowed. Therefore, the concept of a CTS is developed. A CTS is a set of consecutive tokens that arrive at the same discrimination network. The parallel processing of the tokens in a CTS provides Level 2 consistency. To avoid the repeated detection of the CTS, the extended computation unit is used in Level 2 consistency. A technique of detecting the CTSs for a universal type of

discrimination network is developed. We also developed an architecture that consists of three kinds of token queues and two kinds of processes, for the efficient parallel token processing.

Level 1 consistency allows the *untimely joining error* to occur and processes every token that arrives at the system in parallel. Therefore, to make the memory nodes converge when tokens are processed in parallel, we proposed special techniques that use a mechanism called an SLB. The whole technique that makes α nodes converge is called Strategy I, and the whole technique that makes β nodes converge is called Strategy II. The proof of stabilization of α nodes and β nodes is also presented.

Level 0 consistency allows the amount of timing error to extend indefinitely. With this relaxation, a shadow table is not needed for a virtual α node. However, it is shown that another type of duplicate compound token problem exists when the shadow table is not employed. We proposed Strategy III to stabilize β nodes that have virtual α nodes among their descendents. Under the *quiescence assumption*, we proved the stabilization of β nodes.

The type of discrimination network seriously influences the performance and the complexity of the system. We discussed the achievable levels of consistency and the way to achieve them for four kinds of discrimination networks: pure virtual TREAT, TREAT, A–TREAT, and Gator. We hope that this discussion will help system developers when they want to implement the proposed techniques.

CHAPTER 8
CONCLUSION

An active database system monitors the content of the database and automatically executes the associated action when a defined condition holds. A trigger associates an action to be executed with a condition. Triggers provided by many database products execute synchronously. That is, the processing of a trigger is performed within the update transaction that modifies tables on which the trigger is defined. As a result, synchronous triggers make the update transactions become slow.

A system, *TriggerMan*, for trigger processing and view maintenance is under development. TriggerMan is an asynchronous trigger system because its activities are initiated after update transactions to data sources are complete. In a trigger system, a *discrimination network* is built to check the condition of a trigger efficiently. A Gator network is a kind of discrimination network that will be used in TriggerMan.

In the first part of this paper, we propose techniques that will improve the performance of TriggerMan using techniques that do not compromise semantic correctness of trigger processing. The techniques include Gator network dynamic restructuring, efficient processing of large token sets, parallel resource utilization, parallel checking of trigger conditions, and parallel processing of trigger actions and data (memory nodes and base tables).

An optimized Gator network structure is built based on the statistics about the data sources and the costs of functions used in the trigger condition. However, the structure can be sub-optimal over time since those statistics can either be inaccurate or change over time. Therefore, we propose a Gator network dynamic restructuring method that is based on the cost (Gator network priming cost) and the benefit (Gator network performance increase) of restructuring. The cost function of the Gator network optimizer is reused in estimating the performance of a given Gator network. To minimize the cost of restructuring tests, a factor that is specific for each Gator network is incorporated in calculating the schedule of testing for a restructuring.

In propagating multiple tokens that are arriving at a β node simultaneously, the set query approach is beneficial for a large token set, while the tuple query approach is efficient for a small token set. We propose a method that determines the crossover point between the large token sets and the small token sets. Our method uses interpolation on the costs that can be obtained by parsing the output produced by the plan explanation feature of the query optimizer. In relation to that, we are recommending that future database product implementers provide an API function for obtaining the cost of the chosen query-execution-plan. We also propose a strategy that dynamically determines the query-approach type for the tokens arriving at a β node.

The parallel features of three database products have been studied. Based on that study, we propose a method of tuning the parallel features of the host DBMS for the execution of the SQL statements initiated from the TriggerMan system. The parallel execution strategy (the degree of parallelism (DOP) and the parallel scan) of an SQL statement is based on the parallel execution strategies of the nodes that are accessed from

the statement. We list the conditions that make parallel scan beneficial for a Gator network node. Also, we develop the formulas that can be used in determining the DOP of a Gator network node.

The task queue is the central structure that allows concurrent processing. That is, trigger conditions and actions, and tokens wait in the queue to be processed concurrently. The normalized SPI structure is developed to find the triggers with (partially) satisfied selection conditions efficiently. In the SPI structure a triggerID set is associated with a constant in the constant set for an expression signature. The partitioning of a large triggerID set to reduce the granularity of tasks is left as a further study. To achieve data level parallelism, we will utilize the feature provided by the host DBMS and the underlying computer. However, parallel token processing causes problems in the semantic correctness of trigger processing.

In the second part of this paper, we propose four consistency levels of trigger processing to employ parallel token processing in an asynchronous trigger system. The purpose of the consistency level is to pursue maximum performance with controlled and agreed-upon anomalies. The higher the level is, the lower the performance is. Four consistency levels can be summarized as follows: Level 3 provides serial token processing semantics within a trigger; Level 2 allows out-of-order execution of multiple instantiations of the action of a single trigger; Level 1 allows only a limited amount of timing error among the data that initiates a trigger action; and Level 0 guarantees the convergence of the memory nodes of the system.

We then develop techniques that efficiently implement each consistency level. They include the Stability-Lookaside Buffer (SLB) for memory node stabilization,

shadow tables for virtual α nodes, the Concurrent Token Set (CTS) detection and token processing architecture, and the Duplicate-Lookaside Buffer (DLB) for the prevention of duplicate compound tokens.

The primary purpose of an SLB is to stabilize α and β memory nodes. For the efficiency, an SLB needs to be implemented within a *cycle*. Basically, an SLB stores the information about the tokens that modified the memory node in the current cycle. Every SLB is cleared at the end of each cycle. We propose strategies for memory node maintenance. The strategies process tokens with the help of SLBs and stabilize memory nodes. The strategies include Strategy I that stabilizes α nodes, Strategy II that stabilizes β nodes that do not have virtual α node descendents, and Strategy III that stabilizes β nodes that have virtual α node descendents. Strategy I and II provide Level 1 consistency. Strategy III provides Level 0 consistency.

When trigger users can tolerate the out-of-order execution of multiple instantiations of the action of a single trigger, a subset of tokens that arrive at a discrimination network can be processed in parallel. This is the background of a CTS. We propose a technique of detecting the CTS for a universal type of discrimination network. Also, an architecture for parallel token processing is proposed. The set query approach could be integrated with the CTS detection and could further increase system performance. The CTS detection provides Level 2 consistency.

Virtual α nodes cause a *timing error* to expand indefinitely. To limit the timing error to a fixed value, we propose the *shadow table* technique. A shadow table can be shared by multiple virtual α nodes. In relation with a pure virtual α node or with a

shared shadow table, the *duplicate compound token* anomaly can happen. To remove the anomaly, we propose a Duplicate-Lookaside Buffer (DLB) which is a variation of an SLB. A shadow table is needed to provide for the consistency of levels 1, 2, and 3. A DLB needs to be used for each level of consistency.

A discussion about the implementation alternatives of an asynchronous trigger system is given in Section 7.7. This is relevant to system developers who wish to apply the proposed techniques. We believe our proposed techniques will improve overall system performance while creating only controlled and agreed-upon problems in trigger processing.

As further study, we need to develop techniques of making ideal data sources. For example, a technique of creating an ideal data source out of a system like the Sybase Replication Server needs to be developed. The implication of a non-ideal data source needs to be studied. Each and every technique proposed in this paper has not yet been simulated or implemented. Therefore, the simulation and the implementation of the proposed techniques are the things that need to be done next. Especially, the performance comparison of the asynchronous trigger systems that contains the same set of triggers but with different consistency levels would be the most interesting future work.

REFERENCES

[1] Acharya, A. (1994, December). Scalability in production system programs. Ph.D. dissertation, Carnegie Mellon University.

[2] Acharya, A., Tambe, M., & Gupta, A. (1992, July). Implementation of production systems on message-passing computers. *IEEE Transactions on Parallel and Distributed Systems*, 3(4), 477-487.

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

[4] Adelberg, B., Kao, B., & Garcia-Molina, H. (1996, March). Database support for efficiently maintaining derived data. In *Proceedings of EDBT*, 223-240, Avignon, France.

[5] Adiba, M. E., & Lindsay, B. G. (1980, October). Database snapshots. In *Proceedings of the 6th VLDB Conference*, 86-91, Montreal, Canada.

[6] Al-Fayoumi, N. (1998, August). *Design and implementation of a temporal trigger subsystem for the TriggerMan asynchronous rule processor*, Ph.D. dissertation, CISE dept., Univ. of Florida.

[7] Bailey, J., Dong, G., Mohania, M., & SeanWang, X. (1998, July). Incremental view maintenance by base relation tagging in distributed databases. *Distributed and Parallel Databases*, 6(3), 287-309.

[8] Blakeley, J., Coburn, N., & Larson, P. (1989, September). Updated Derived Relations: Detecting Irrelevant and Autonomously Computable Updates, *ACM TODS*, 14(3), 369-400.

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

[10] Bodorik, P., Riordon, J. S., & Pyra, J. S. (1992, June). Deciding to correct distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 4(3), 253-265.

[11] Brownston, L., Farrell, R., Kant, E., & Martin, N. (1985). *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming*. Reading, MA: Addison-Wesley.

[12] Butler, P. L., Allen, J. D., & Bouldin, D. W. (1988). Parallel architecture for OPS5. In *Proceedings of the 15th International Symposium on Computer Architecture,* 452-457.

[13] Carnes, C. (1998). *A Flexible Data Source Architecture for an Asynchronous Trigger Processor*, Ph.D. dissertation, CISE dept., Univ. of Florida. (in preparation)

[14] Ceri, S., & Widom, J. (1991, September). Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 577-589, Barcelona, Spain.

[15] Chase, R. B., & Aquilano, N. J. (1989). *Production and Operations Management: A Life Cycle Approach*. (5th ed.). Homewood, IL: IRWIN.

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

[17] Christodoulakis, S. (1984, June). Implication of certain assumptions in database performance evaluation. *ACM TODS,* 9(2), 173-186.

[18] Colby, L. S., Griffin, T., Libkin, L., Mumick, I. S., & Trickey, H. (1996, June). Algorithms for deferred view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 469-480.

[19] Date, C. J. (1995). *An Introduction to Database Systems*. (6th ed.). Reading, MA: Addison-Wesley.

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

[21] DeWitt, D. J., Ghandeharizadeh, S., Schneider, D. A., Bricker, A., Hsiao, H., & Rasmussen, R. (1990, March). The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 44-62.

[22] Elmasri, R., & Navathe, S. B. (1994). *Fundamentals of database systems*. (2nd ed.). Redwood City, CA: Benjamin/Cummings.

[23] Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19, 17-37.

[24] Gehani, N., & Jagadish, H. (1991, September). Ode as an Active database: Constraints and triggers. In *Proceedings of the 17 th International Conference on Very Large Databases*, 327-336, Barcelona, Catalonia, Spain.

[25] Gehani, N., & Jagadish, H. (1992). Active database facilities in Ode. *Data Engineering Bulletin*, 15(1-4), 19-22.

[26] Geier, J. T. (1996). *Network Reengineering: the New Technical Imperative*. New York: McGraw-Hill.

[27] Gray, J. N., Lorie, R. A., Putzolu, G. R., & Traiger, I. L. (1994). Granularity of locks and degrees of consistency in a shared data base. In M. Stonebraker (Eds.), *Readings in database systems* (2nd ed., pp. 181-208). San Francisco, CA: Morgan Kaufmann.

[28] Gupta, A. (1987). *Parallelism in production systems*. Los Altos, CA: Morgan-Kaufmann.

[29] Gupta, A., & Forgy, C. (1983, December). *Measurements on production systems*, (Tech. Report No. CMU-CS-83-167). Computer Science Dept., Carnegie Mellon University.

[30] Gupta, A., Forgy, C, Kalp, D., Newell, A., & Tambe, M. (1987). *Result of Parallel implementation of OPS5 on the Encore multiprocessor*, (Tech. Report No. CMU-CS-87-146). Computer Science Dept., Carnegie Mellon University.

[31] Gupta, A., Forgy, C., & Newell, A. (1989, May). High-speed implementations of rule-based systems. *ACM TOCS*, 7(2), 119-146.

[32] Gupta, A., & Mumick, I. S. (1995, June). Maintenance of materialized views: problems, techniques and applications. *IEEE Data Engineering Bulletin*, 18(2), Special Issue on Materialized Views and Data Warehousing.

[33] Hanson, E. N. (1992, June). Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 49-58.

[34] Hanson, E. N. (1993, August). Gator: A generalized discrimination network for production rule matching. In *Proceedings of the IJCAI Workshop on Production Systems and Their Innovative Applications*.

[35] Hanson, E. N. (1996, February). The design and implementation of the Ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), 157-172.

[36] Hanson, E. N., Al-Fayoumi, N., Carnes, C., Kandil, M., Liu, H., Lu, M., Park, J., & Vernon, A. (1997, December 15). *TriggerMan: An Asynchronous Trigger Processor as an Extension to an Object-Relational DBMS*, (Tech. Report No. 97-024). CISE Dept., University of Florida. Available: http://www.cise.ufl.edu/research/ tech-reports [January 1998].

[37] Hanson, E. N., Bodagala, S., & Chadaga, U. (1997, November). *Optimized Trigger Condition Testing in Ariel Using Gator Networks*, (Tech. Report No. 97-021). CISE

Dept., University of Florida. Available: http://www.cise.ufl.edu/research/tech-reports [15 December 1997].

[38] Hanson, E. N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S., Park, J., Vernon, A. (1999, March). *Scalable Trigger Processing*, In *Proceedings of the 15th International Conference on Data Engineering*, 266-275, Sydney, Australia.

[39] Hanson, E. N., & Hasan, M. S. (1993, December). Gator: An optimized discrimination network for active database rule condition testing. (Tech. Report No. TR93-036). CISE Dept., University of Florida. Available: http://www.cise.ufl.edu/research/tech-reports.

[40] Hanson, E. N., & Johnson, T. (1996). Selection predicate indexing for active databases using interval skip lists. *Information Systems,* 21(3), 269-298.

[41] Horowitz, E., & Sahni, S. (1978). *Fundamentals of computer algorithms*. Rockville, MD: Computer Science Press.

[42] Huyn, N. (1997). Multiple-view self-maintenance in data warehousing environment. In *Proceedings of the 23rd VLDB Conference*, 26-35, Athens, Greece.

[43] IBM Corp. (1997). *DB2 SQL Reference* [Online documentation]. Available: http:// www.software.ibm. com/cgi-bin/db2www/library/pubs.d2w/report#UDBPUBS [12 January 1998].

[44] IBM Corp. (1996). *DB2 Parallel Edition Administration Guide and Reference* (Version 1.2) [Online documentation]. Available: http://www.software.ibm.com/cgi-bin/db2www/library/pubs.d2w/report# UDBPUBS [12 January 1998].

[45] Informix Software, Inc. (1997, March). *INFORMIX-Universal Server, Informix Guide to SQL: Syntax* (Version 9.1) [Online documentation]. Available: http://www. informix.com/answers/oldsite/answers/ english/912iusnt.htm [27 January 1998].

[46] Informix Software, Inc. (1997, March). *INFORMIX-Universal Server: Administrator's Guide* (Version 9.1) [Online documentation]. Available: http://www.informix.com/answers/oldsite/answers/ english/912iusnt.htm [15 January 1998].

[47] Ioannidis, Y., & Christodoulakis, S. (1991, May). On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD Conference on the Management of Data*, 268-277, Denver, CO.

[48] Isakowitz, T., Bieber, M. P., & Vitali, F. (1998, July). Web information systems: introduction. *Commun. ACM,* 41(7), 78-80.

[49] Ishida, T. (1991, March). Parallel rule firing in production systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(1), 11-17.

[50] Ishida, T. (1994, August). An optimization algorithm for production systems. *IEEE Transactions on Knowledge and Data Engineering*, 6(4), 549-557.

[51] Jagadish, H.V., Mumick, I. S., & Silberschatz, A. (1995) View maintenance issues for the chronicle data model. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems 1995*, 113-124, ACM, New York, NY.

[52] Jagadish, H. V., Narayan, P. P. S., Seshadri, S., Sudarshan, S., & Kanneganti, R. (1997, August). Incremental organization for data recording and warehousing. In *Proceedings of the 23rd VLDB Conference*, 16-25, Athens, Greece.

[53] Jarke, M., & Koch, J. (1984, June). Query optimization in database systems. *ACM Comp Surv*., 16(2).

[54] Johnston, W. (1997, September). Rationale and strategy for a 21st century scientific computing architecture: the case for using commercial symmetric multiprocessors as supercomputers. *International Journal of High Speed Computing*, 9(3), 191-222.

[55] Kabra, N., & DeWitt, D. J. (1998, June). Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 27(2), 106-117.

[56] Kandil, M., (1998). *Selection Predicate Placement in Database Discrimination Networks*, Ph.D. dissertation, CISE dept., Univ. of Florida.

[57] Konyala, M. K. (1998). *Predicate Indexing in TriggerMan,* MS thesis, CISE dept., Univ. of Florida.

[58] Mannino, M. V., Chu, P., & Sager, T. (1988, September). Statistical profile estimation in database systems. *ACM Comput. Surv.*, 20(3), 191-221.

[59] McCarthy, D. R., & Dayal, U. (1994). The Architecture of an Active Data Base Management System. In M. Stonebraker (Eds.), *Readings in database systems* (2nd ed., pp. 373-382). San Francisco, CA: Morgan Kaufmann.

[60] Miranker, D. P. (1987, August). TREAT: a better match algorithm for AI production systems. In *Proceedings AAAI National Conference on Artificial Intelligence*, 42-47.

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

[62] Negri, M., Pelagatti, S., & Sbatella, L. (1991, September). Formal Semantics of SQL Queries, *ACM TODS*, 16(3), 513-534.

[63] O'Neil, P. (1994). *Database: Principles, Programming, and Performance*. San Francisco, CA: Morgan Kaufmann.

[64] Oracle Corp. (1997, June). *Oracle8 Server Concepts* (Release 8.0) [Online documentation]. Available: http://www.oracle.com.sg/support/products/o8s/nt/html/doc.html [7 January 1998].

[65] Patterson, D. A., & Hennessy, J. L. (1990). *Computer architecture: a quantitative approach*. San Mateo, CA: Morgan Kaufmann.

[66] Ross, K. A., Srivastava, D., & Sudarshan, S. (1996, June). Materialized view maintenance and integrity constraint checking: trading space for time. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 447-458.

[67] Roussopoulos, N. (1991, September). An Incremental Access Method for View-Cache: Concept, Algorithms, and Cost Analysis. *ACM TODS*, 16(3), 535-563.

[68] Sivazlian, B.D., & Stanfel, L.E. (1974). *Analysis of Systems in Operations Research.* Englewood Cliffs, NJ: Prentice-Hall.

[69] Stonebraker, M., Jhingran, A., Goh, J., & Potamianos, S. (1994). On Rules, Procedures, Caching and Views in Data Base Systems. In M. Stonebraker (Eds.), *Readings in database systems* (2nd ed., pp. 363-372). San Francisco, CA: Morgan Kaufmann.

[70] Stonebraker, M., Rowe, L., & Hirohama, M. (1990, March). The implementation of POSTGRESS. *IEEE Transactions on Knowledge and Data Engineering*, 2(7), 125-142.

[71] Sybase, Inc. (1996). *Sybase Adaptive Server 11.5: EnterpriseTransact-SQL User's Guide* [Online documentation]. Available: http://sybooks.sybase.com/dynaweb/group2/asg1150e/sqlug/@Generic__BookView [5 March 1998].

[72] Wang, Y. W., & Hanson, E.N. (1992, February). A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proceedings of IEEE Data Eng. Conf*., 88-97.

[73] Widom, J. (1996, August). Starburst active database rule system. *In IEEE Transactions on Knowledge and Data Engineering*, 8(4), 583-595.

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

[75] Yao, S.B. (1977, April). Approximating block accesses in database organizations. Communications *of the ACM*, 20(4), 260-261.

[76] Zhuge, Y., Garcia-Molina, H., Hammer, J., & Widom, J. (1995, May). View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 316-327, San Jose, CA.

BIOGRAPHICAL SKETCH

Jongbum (J.B.) Park is from Korea. He graduated from the Korea Air-Force Academy in 1984. He earned his bachelor's degree in computer science at Seoul National University in 1988. He received the master's degree in computer science at Pohang University of Science and Technology in 1991. Before starting his Ph.D. studies, he worked as a faculty member at the Korea Air-Force Academy. Since 1994, he has been working towards the doctoral degree in computer engineering (CISE) at the University of Florida.