A DECOMPOSITION-BASED APPROACH TO JOIN TRIGGER PROCESSING IN TRIGGERMAN

By

SASI KUMAR PARTHASARATHY

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1999

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Eric Hanson, for giving me an opportunity to work on this challenging project and also for providing continuous guidance, advice, and support throughout the course of my work.

I thank Dr. Herman Lam and Dr. Sharma Chakravarthy for serving on my supervisory committee and for their careful perusal of this thesis. I would like to specially thank Dr. Y. C. Randy Chow for accepting my request to be a substitute member on my supervisory committee. I would also like to thank Ms. Sharon Grant for maintaining a great research environment at the Database Systems Research and Development Center.

I am thankful to Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, J. B. Park and Albert Vernon for their invaluable help and the fruitful discussions that we had during the course of this work.

On a more personal note, I would like to thank my family whose love, support, and constant encouragement was of great importance through this work.

TABLE OF CONTENTS

LIST OF FIGURES

Page

Abstract of the Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A DECOMPOSITION-BASED APPROACH TO JOIN TRIGGER PROCESSING IN TRIGGERMAN

By

Sasi Kumar Parthasarathy

December 1999

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

The synchronous database trigger systems of today have limited scalability and extensibility. They can hardly handle a few triggers on a single data source table. Also synchronous trigger processing system lengthens the response time of update transactions. With the advent of the Internet, it has become necessary that trigger systems are scalable and the users could create any number of triggers through web interface without any limitations. TriggerMan is one such scalable and extensible asynchronous trigger processing system that offsets both the disadvantages mentioned above. This thesis presents a novel approach to efficient condition testing for multi-table join triggers in TriggerMan.

A decomposition-based approach is used for join trigger processing in TriggerMan. The Treat discrimination network was chosen for condition testing and is created for each trigger during compile time. It is supplemented with query templates (consisting of FROM CLAUSE, TARGET CLAUSE and WHERE CLAUSE) that are built at compile time for each table appearing in the trigger definition. This makes it easier, simpler and faster to build, using these templates, the queries to check for join trigger conditions at run time. The method of tuple substitution is used while building the query from the templates at trigger condition testing time.

CHAPTER 1
INTRODUCTION

A database management system extended with trigger processing capabilities is called an active database management system. Conventional database systems store data passively and perform only actions explicitly requested by a user transaction. An active database system, in addition to performing the activities of a normal database management system has the ability to execute actions automatically when specified events of interest occur [Sil98]. Because of their wide applications, active database systems have gained considerable attention among the database research community in the recent past. Many prototype active database systems have been built [Cha89, Han96, Geh91] and also many commercial database systems support a rule processing facility. Even though many of the database products support active database features, they are very much limited in their capability i.e., at the most one to a few dozen triggers per table can be defined on each update event. This is because current database systems process triggers synchronously. Synchronous trigger processing lengthens the response time of update transactions, because the transaction cannot commit until all the trigger conditions defined on the updated tables are tested and actions fired. The response time of update transaction is thus directly proportional to the number of triggers, which limits the number of triggers that can be defined on a table.

In the case of assertions synchronous triggers are indispensable. But in most other cases, delaying the trigger processing to until after the triggering transaction commits (i.e. asynchronous trigger processing) should serve the purpose. This is especially true in the case of alerters. In an asynchronous trigger system, the only work expended during a transaction is capturing the updates for later processing [Han98]. The time spent in capturing updates is a constant and totally unrelated to the number of triggers defined on the tables involved in the transaction. Thus, with asynchronous trigger processing, it is possible to increase the number of triggers defined on a table to a very large number without affecting the response time of update transactions.

After the transaction completes, each of the updated tuples captured (known as tokens) are individually tested against the conditions of the set of triggers using a predicate indexing technique [Han98] and actions fired accordingly. This process of testing all the trigger conditions can be done sequentially or in parallel.

Even though lots of advances have taken place in active database research, they are yet to show up in the database products because of the complexity involved in implementation. A huge speedup can be achieved by means of an efficient discrimination network for evaluating predicates. Different approaches to predicate indexing have been proposed for efficient condition testing. They range from simple sequential search to complex multi-dimensional indexing [Hans97]. New data structures such as the Interval Skip List [Han96b] and Interval Binary Tree [Han90] [Han94] have been proposed for indexing the predicates. However they are very much limited in their scope. Some of them are not scalable and some are not extensible for User Defined Data types. The Selection Predicate Index [Han99], is one such discrimination network that is scalable,

extendable and also gives a considerable speedup in the processing time per updated token.

The strategy of using a Selection Predicate Index as a discrimination network to evaluate selection predicates works well for all types of triggers, especially with single table triggers because the entire condition evaluation on an update token takes place in main memory i.e., without accessing the database. In the case of join triggers, there is a relative increase in time spent on processing a token, due to a possible table-access involved during condition evaluation of the trigger. In the case of join triggers, when an update token of a table arrives, a part of the condition evaluation involves performing a join.

In this dissertation, we are going to discuss an efficient way of evaluating the multi-table triggers using an approach based on trigger condition decomposition. Let us take a real life example of a join trigger. Consider two database tables namely Employee and Department given below:

Employee (empno, empname, salary, sex, address, city, state, deptno)

Department (deptno, deptname, designation)

Lets consider a trigger as follows: ‘When the salary of any employee who is in Gainesville and who works for the Finance department goes above $5000, inform manager Bob’

The E-C-A model is the most widely used model to specify rules in a database environment. The format of an ECA rule is given as:

on event

if condition

then action

The trigger definition for the above mentioned trigger satisfying the ECA rule is as follows:

Create trigger trigger1

From employee, department

On event

{update/insert to employee, department}

If condition

{When employee.city = ‘Gainesville’ and

employee.salary >= 5000

employee.empno = department.empno and

department.deptname = ‘Finance’}

then action

{Do “Inform manager Bob”}

This trigger is tested for an event that can be either an insert or update to either the employee or the department table. Whenever the event of interest occurs, the conditions are tested as follows:

Whenever there is an update/insert to employee table, the tuple is checked for the city = ‘Gainesville’ and salary >= $5000. If it satisfies these conditions, then a query is formulated to select all the tuples from the department database which satisfies deptname = ‘Finance’. If the select statement returns one or more tuples, then the action is executed

which is to inform Bob. This is how a multi-table join query is tested for events of interest.

This chapter talked about the evolution of active databases, their emerging importance and how discrimination networks like the Selection Predicate Index can speed up condition evaluation. In Chapter 2 we will discuss the background of the query decomposition approach with reference to the Ingres DBMS. In Chapter 3 we will discuss the features of TriggerMan, the asynchronous trigger processing system in which the decomposition based approach to join trigger processing has been implemented, tested and whose performance has been measured. Chapter 4 discusses the details of the decomposition-based approach to join trigger processing and its implementation, and Chapter 5 discusses the finer points of this approach.

CHAPTER 2
OVERVIEW OF THE TRIGGERMAN

2.1 Introduction

Although many active database systems like Ariel [Han96a], POSTGRES [Sto91], HiPAC [Cha89], and Ode [Geh91] have been developed, they do not provide the scalability feature and have very limited, if any extensibility support. Among them, Ariel and POSTGRES do have selection predicate indexes (SPI) to evaluate predicate conditions, but these are not extensible.

TriggerMan is a scalable and extensible asynchronous trigger processor. It is extensible in the sense that the triggers in TriggerMan can have user defined data types (UDTs) and the system is scalable as it works well even for millions of triggers for certain applications. The scalability of the system can be attributed mainly to its asynchronous trigger processing mechanism and an efficient predicate indexing strategy. As the number of triggers in a system increases, more time is spent on testing each updated tuple against the trigger conditions. Making this testing process asynchronous alleviates the problem of delays in update transactions, which is an inherent problem of synchronous trigger processing. TriggerMan implements the Selection Predicate Indexing strategy [Han99] to evaluate the predicates which results in a large speedup, thus reducing the processing time per updated tuple.

TriggerMan is developed as a **DataBlade,** an extension module for Informix with Universal Data Option (UDO). DataBlade modules run in the Informix server's address space. TriggerMan is designed mostly as a platform independent module so that it can be ported to other ORDBMS platforms in the future.

2.2 TriggerMan Architecture

The architecture of TriggerMan is shown in Figure 2.1. The main components of the architecture are explained as follows:

- The **TriggerMan Module** resides and runs in the address space of the DBMS server.
- The **Data Source applications** are responsible for capturing updates on local and remote data sources and putting them in an update queue table. An update descriptor is a tuple consisting of new and old values of a tuple and the details of operation performed on the tuple by an update transaction on the data source.
- The **TriggerMan drivers** basically perform two functions periodically. One of them is to gather the update descriptors and add them as tasks to the task queue. The task queue is a to-do list of tasks for the trigger processor. The other function is to read and process jobs from the task queue for condition evaluation and action execution. Multiple drivers run concurrently to perform these tasks in parallel.
- The **TriggerMan console** is an application program that allows the user to initialize the system, create triggers, drop triggers, shut down the system etc.

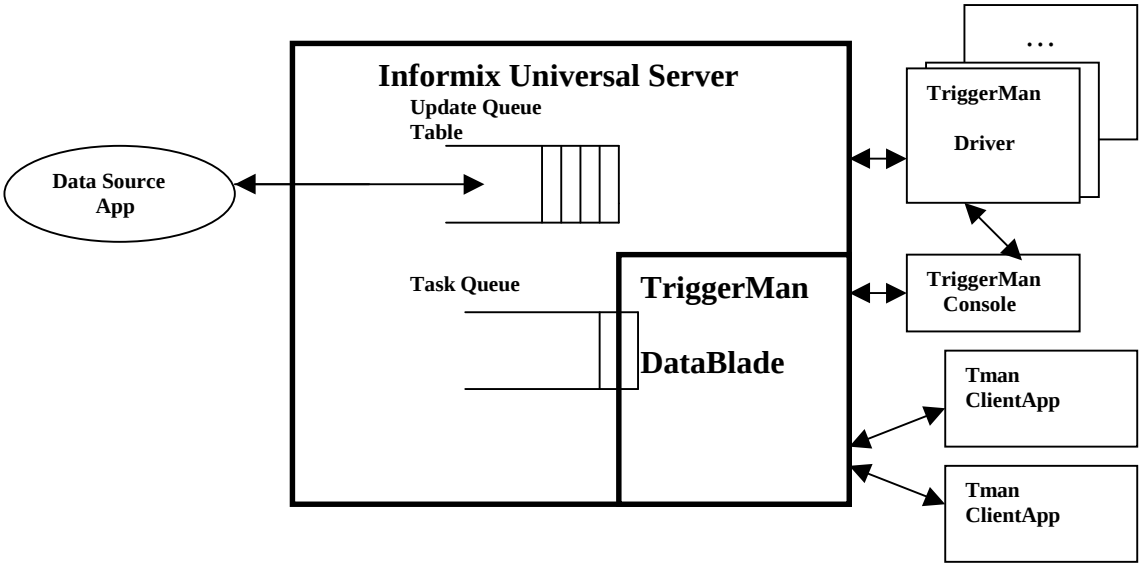

Figure 2.1: TriggerMan architecture

2.3 TriggerMan Command Language

TriggerMan uses an SQL-like syntax for the command language. The command language supports the following:

- Defining data sources
- Defining triggers

2.4 The Selection Predicate Indexing Scheme

One of the two major functions of a TriggerMan driver program shown in Figure 2.1 is to evaluate the predicates for an updated tuple against each of the trigger

definitions. The Selection Predicate Indexing scheme [Han98] as shown in Figure 2.2 employed by TriggerMan to evaluate the predicates gives a tremendous speedup when compared to schemes used by conventional databases.

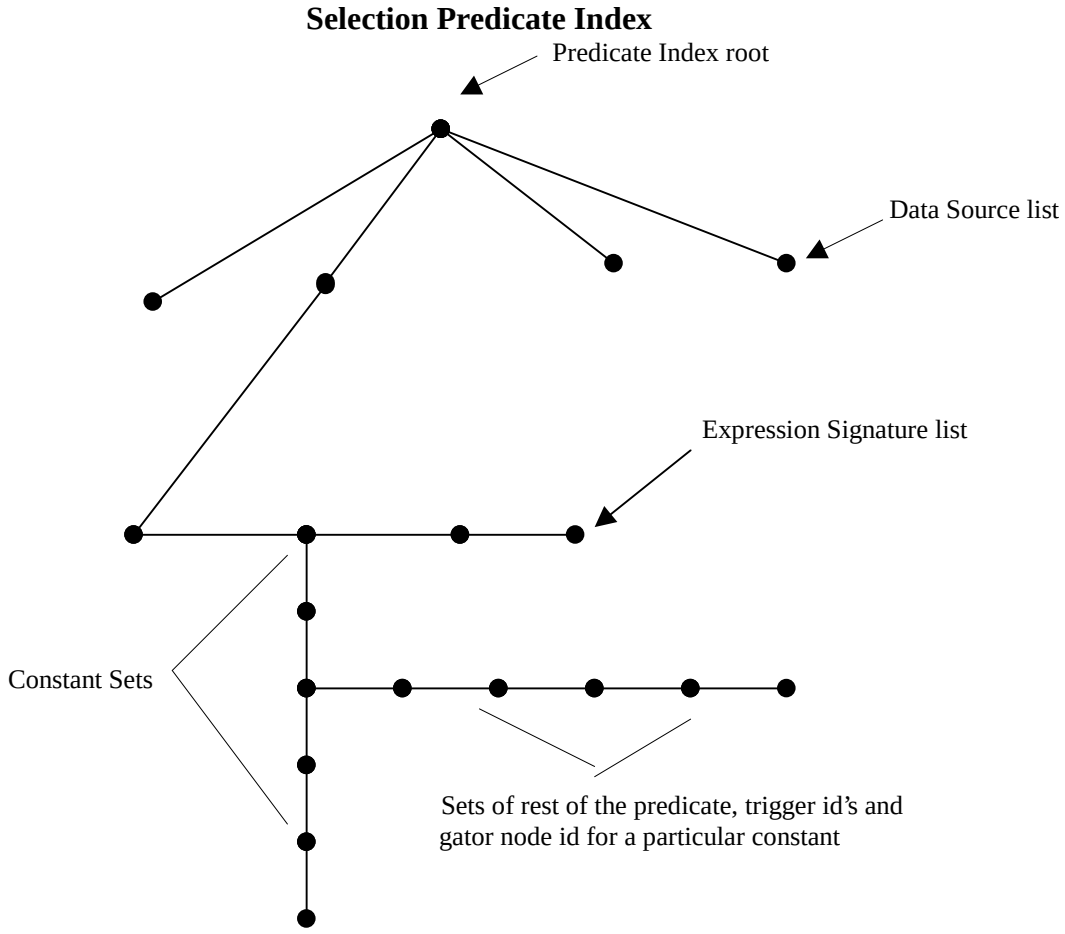

Figure 2.2: Predicate index architecture

The basic premise of the Selection Predicate Indexing strategy shown in Figure 2.2 is that in a system having a large number of triggers, there is always a fair chance that groups of triggers have a part of their predicate condition looking similar. For example, a majority of triggers defined by a stockbroker will be alerters where the predicates check for stock prices respectively going above or below a certain value of interest.

All the data sources defined in TriggerMan have a node at the first level below the root. When a trigger is defined, its predicate is converted to Conjunctive Normal Form (CNF), and the most selective conjunct qualifies to be the expression signature for that trigger. If the same signature already exists, then the constants of this conjunct are plugged into the constant list under that signature and nodes containing the rest of the predicate and the id of the trigger along with the gator node id are placed under this constant. If the same signature does not already exist then a new signature node is created and the same steps followed.

For example let us consider the following two single table triggers having the same expression signature:

```
create Trigger quota_check_bill
        from account_info
        where (name = "Bill") and (quota > 5000000)
        do email("Administrator Murray", "Alert Bill that he his over-limit");
create Trigger quota_check_bob
        from account_info
        where (name = "Bob") and (quota >= 4000000)
        do email("Administrator Murray", "Alert Bob that he is nearing his limit");
```

The trigger processor isolates the predicate "account_info.name = constant" as the expression signature for both the triggers because it is the most selective. It adds this signature into the list of expression signatures under the data source "account_info" (if it does not already exist). The constants "Bill" and "Bob" are inserted into the list of constants (if they do not already exist). Finally, a node for the ids' of the triggers and the remaining part of the predicate are inserted under these constants.

Now the updated tuple of the account_info relation pulled out from the task queue by the TriggerMan driver is made to pass through this discrimination network. The attribute values within this tuple are evaluated against the constants within each expression signature. For every constant node whose expression evaluates to true, we proceed with testing the rest of the predicate and then firing the trigger action if necessary.

The predicate for single table triggers can be evaluated completely within this main memory discrimination network.

Processing is direct for single table triggers as discussed above. However for join triggers which may involve one or more tables, when the selection predicate evaluates to true, the remaining tables involved in the trigger have to be checked. The approach we have developed for checking join triggers is based on a novel application of tuple substitution, in combination with the use of a special query template for each tuple variable in the trigger condition that is constructed at compile time for every trigger definition. The rest of the chapters discuss an efficient way we have developed to perform this operation and then show performance measurements of the join processing system.

CHAPTER 3
RELATED WORK

3.1 Introduction to Ingres

An earlier system that had some influence on the design we have chosen for join trigger processing in TriggerMan is INGRES (Interactive Graphics and Retrieval System). Ingres [Sto76] is a relational database system that is implemented on top of the UNIX operating system. INGRES supports QUEL (Query Language) which is its primary query language and for purposes of providing a customized user interface also supports EQUEL (Embedded QUEL). EQUEL is QUEL embedded in the programming language C.

INGRES has been implemented as a layered process structure. The commands are tokenized and let flow from one process to the next one serially. Decomp is one of the processes in INGRES which is involved in decomposition of queries involving more than one variable into sequences of one variable queries. It achieves this by accumulating the partial results on the way until the query is evaluated.

Because INGRES allows queries that are defined on the cross product of one or more relations, efficient execution of the query evaluation is given the utmost importance. The objective can be achieved by searching as small a portion of the appropriate crossproduct space as possible.

3.2 Query Evaluation Using Decomp

Decomp follows a technique of tuple substitution to reduce a query to fewer variables. A variable, chosen out of possibly many is selected for substitution and for each such tuple, the values of domains in that relation are substituted into the query. In the resulting modified query, all previous references to the substituted variable have now been replaced by constant values and the query has thus been reduced to one less variable. Decomposition is recursively repeated on the modified query until it is reduced to only one variable at which point the built-in OVQP (One Variable Query Processor) is called on that one variable query. Query evaluation by this method has the effect of reducing the size of the relation over which the sub-query (without this substituted variable) ranges by choosing a subset of values for the sub-query.

Let us trace out the decomposition technique using an example query. Consider two database tables namely Employee and Department as given below:

Employee (empno, empname, salary, manager, sex, age, city, state, deptno)

Department (deptno, deptname, designation)

The Query 1 is as follows: ‘Request for all the employees who are from Gainesville and who work for the Finance department.’

The equivalent QUEL query for the above request is as follows:

Range of emp is Employee

Range of dept is Department

Retrieve (Emp.name)

Where emp.city = “Gainesville” and

emp.deptno = dept.deptno and

dept.deptname = “Finance”

The graphical representation of query 1 before applying decomposition to evaluate it is given in Figure 3.1.

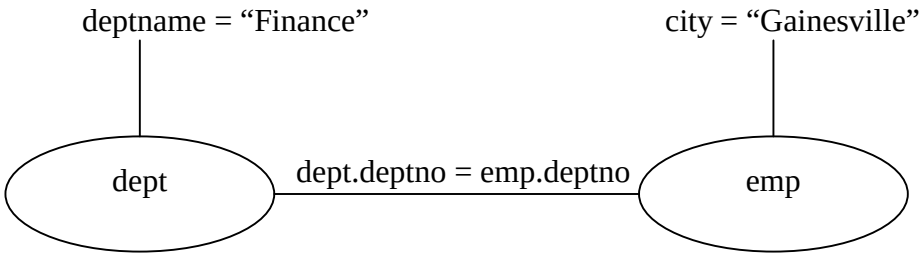

Figure 3.1: Graphical representation of query 1 before decomposition

The of steps in the recursive decomposition to evaluate this query is as follows:

1. Retrieve into temp1 all the employees from the employee database by calling the OVQP to process a one variable query like

   Select * from employee where employee.city = “Gainesville”

   Now temp1 has the tuples that have been selected as a result of the above query. As a result, we have effectively eliminated running the join condition on the whole employee table. Instead it can be run on the temporary view temp1. The effect can be fully realized when the selection predicate on the

employee table is highly selective. The graphical representation of query 1 after step 1 is given in Figure 3.2.

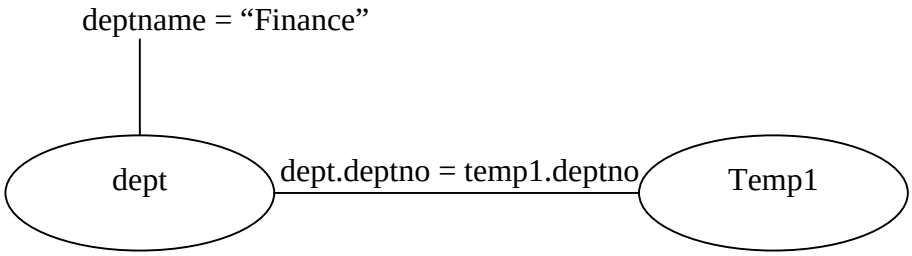

Figure 3.2: Graphical representation of query 1 after step 1

2. Now it is time for the second level of substitution. Instead of running the join on the employee database, it is now run on a subset of the employee database containing the employees from Gainesville stored in temp1 as shown in Figure 3.2. The result of this join is nothing but a set of constant values. We can also note that the number of tuple variables has been effectively reduced from two to one.
3. As a result of repeated tuple substitution, the one variable query looks as shown in Figure 3.3. The built-in one variable query processor is called now to process this one variable query.

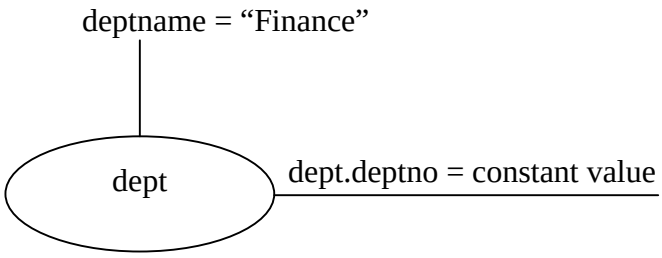

Figure 3.3: Graphical representation of query 1 after step 2

The algorithm thus decomposes the original query into a sequence of one variable queries with the results merged appropriately.

3.3 Conclusion

TriggerMan uses something similar to the tuple substitution method of decomp technique to process join triggers. The remaining chapters discuss the way join triggers are processed in TriggerMan using the query template during compile time. They also talk about how the tuple substitution technique plays a crucial role during run time in checking these triggers against the updated tuples.

CHAPTER 4
DECOMPOSITION BASED JOIN TRIGGER PROCESSING

4.1 Introduction

The previous chapter presented the strategy of query processing by decomposition as implemented in Ingres. This chapter discusses how the technique of query template formation during compile time coupled with a variation of decomposition during run time is used in TriggerMan to process join triggers.

4.2 Overview of Treat Network

Discrimination networks are tree structures with stored or virtual nodes constructed to test the predicate conditions of rules efficiently. They can be classified into Rete [For82], Treat [Mir87] and Gator [Han92] networks based on the tree structure. Before the rule condition is compiled into a discrimination network, it can be represented by a Rule Condition Graph (RCG) which has a node for each tuple variable and an edge for each join condition.

TriggerMan builds a version of a Treat network constructed from its RCG at compile time for each and every trigger defined. Let us discuss the Treat network a little bit before going into the details of trigger processing at compile time. Consider two database tables namely Relation1 (a, b, c) and Relation2 (a, b, c) and a sample trigger defined on these tables using the TriggerMan command language:

Create Trigger foo

from Relation1 as R1, Relation2 as R2

when R1.a = constant1 and

R1.b = R2.b and

R2.c = constant2

do echo ”foo fired”;

The Treat network for the above trigger is shown in Figure 4.1.

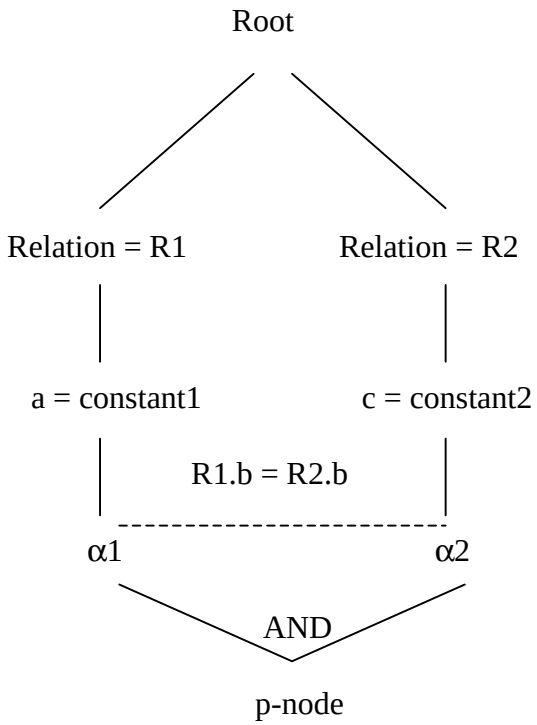

Figure 4.1: Treat network for trigger foo

The essential components of a Treat network shown in Figure 4.1 is as follows:

- **Root node:** The root node is always at the top of any discrimination network.
- **P-node:** The results matching any trigger condition after checking for all selection and join predicates are placed here. There is always only one p-node per trigger. It is always at the bottom of any discrimination network.
- **Constant node:** There is one constant node for each tuple variable in the rule condition. These nodes check for selection conditions.
- **α-node:** The results of the selection condition as tested by the constant node flow down to the alpha nodes. There are two types of alpha nodes namely stored alpha and virtual alpha. Unlike stored alpha nodes that store the result of any selection condition, virtual alpha nodes just lets them flow through. So with virtual alpha nodes, we need to evaluate the selection predicate whenever we test for a trigger whereas with stored alpha nodes, we can use the same results over and over again until the table has been updated. But we do have a problem of maintaining consistent information across the alpha nodes and the database.

4.3 Trigger Processing at Compile Time

Trigger processing activities in TriggerMan can be categorized into compile time functions and run time functions. The compile time processing includes executing the “create trigger” statement and creating the necessary main memory data structures such as the selection predicate index and the modified Treat network for this trigger and update

the trigger catalogs to store the information about this trigger. The run time processing occurs whenever the TriggerMan driver program checks an update descriptor.

When a Create Trigger statement is processed, a number of steps must be performed to update the trigger system catalogs and main memory data structures. Let us illustrate how a multi-table join trigger is processed by taking a sample trigger defined on the tables R1 (a, b, c), R2 (a, b, c) and R3 (a, b, c):

```
Treate Trigger foo_bar
from R1, R2, R3
when R1.a = val1 and
        R1.b = R2.b and
        R2.a = val2 and
        R2.c = R3.c and
        R3.a = val3
do echo ”foo_bar fired”;
```

The steps to be followed to process a Create Trigger statement are as follows:

- Parse the trigger statement and validate it.
- Convert the WHEN clause to CNF and group the conjuncts by the distinct sets of tuple variables they refer to. In TriggerMan terminology, the conjunction of all the selection predicates defined on a single tuple variable is called the Decorated Selection Predicate.
- Form the Rule Condition Graph (RCG) for this trigger as shown in Figure 4.2.

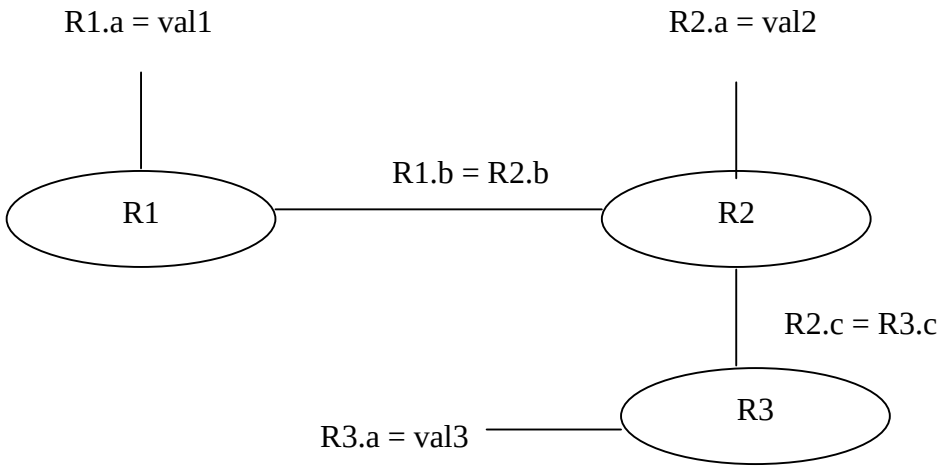

Figure 4.2: RCG for trigger foo_bar

- Form the virtual Treat network for this trigger. The Treat network for foo_bar is shown in Figure 4.3.

As we see, there seems to be a few changes from the TriggerMan Treat in figure 4.3 when compared to the conventional Treat network in figure 4.1. The TriggerMan Treat does not have a conventional root node. Also, the TriggerMan Treat is a pure virtual Treat network with no stored alpha nodes. We discuss why we chose virtual Treat

network over the gator network and why we use virtual alpha nodes rather than stored alpha nodes in the next chapter.

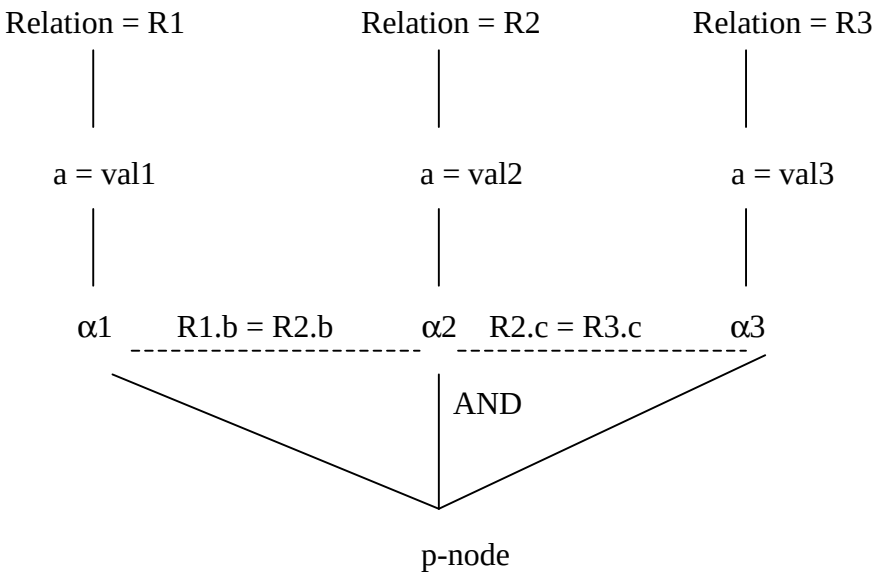

Figure 4.3: Treat network for trigger foo_bar

The novel feature about join query processing is that TriggerMan builds a query template during compile time. Each and every alpha node has a unique ID assigned by the system during the Treat network construction. Each and every alpha node has a query template associated with it. The purpose of a query template is to simplify the job of constructing a select statement to test for the join trigger. The concept of query template

leads the way for using decomposition for query processing. A query template basically contains the three clauses to frame a select statement namely the FROM clause containing the tuple variables involved in the query, the TARGET clause containing the attributes to be selected and the WHEN clause containing the conditions to be satisfied.

For example, let us see how the query template for alpha node α1 looks.

FROM clause: R2, R3

TARGET clause: all attribute values from an R1 tuple, R2. *, R3. *

WHEN clause: R1.b = R2.b and

R2.a = val2 and

R3.c = R2.c and

R3.a = val3

Now that we have all the necessary information to frame a query to check for the join trigger, we just plug in the values from an update descriptor for all the underlined attributes in the WHEN clause. Let us assume a tuple t1 (const1, const2, const3) is inserted into table R1. If this tuple descriptor satisfies the selection predicate R1.a = val1 (when val1 = const1), then we have got to check whether this trigger has to be fired. We construct a select statement using the FROM clause, TARGET clause and the WHEN clause we have and plug in the values for R1.b from the tuple t1. The select statement looks like:

Select t1.*, R2.*, R3.*

From R2, R3

Where const1 = R2.b and

R2.a = val2 and

R2.c = R3.c and

R2.a = val3

The trigger action is executed for every tuple selected by this select statement and is not fired if nothing is selected.

- We now return to a discussion of how the trigger is compiled. After building the Treat network, for each tuple variable, determine the indexable predicate to be the expression signature. Let us assume that (C-11 OR C-22 OR … OR C-1N) AND … AND (C-K1 OR C-K2 OR … OR C-KN) {where all clauses C-IJ appearing in the predicate refer to the same variable} is the decorated selection predicate for tuple variable t. Furthermore, each such clause is an atomic expression that does not contain any boolean operators other than possibly the NOT operator. However, a single clause may contain constants. To find the indexable predicate and the rest of the predicate, check the predicate list for predicates of type variable name = constant.

Case 1: If one or more is found,

- The expression signature is of the form (variable1 = constant1 AND variable2 = constant2 AND … AND variable-n = constant-n) where variable1, variable2 … variable-n are the attributes of particular tuple variable for which the expression signature is defined.
- The rest of the predicate is the conjunction of the remaining predicates.

Case 2: If none are found,

- The conjunction of all the predicates as a whole is the expression signature.
- The rest of the predicate is null.

For the trigger foo_bar, the indexable predicate that is selected as the expression signature and the rest of the predicate is as follows:

1. For tuple variable R1, the indexable predicate is R1.a = constant and the rest of the predicate is null.
2. Similarly for R2, the indexable predicate is R2.a = constant and the rest of the predicate is null.

These steps signify the creation of a trigger after executing the Create Trigger statement.

4.4 Trigger Processing at Run Time

Trigger processing at run time consists of all operations from the moment the TriggerMan driver starts processing an update descriptor. Let us trace the steps when the TriggerMan driver processes an insert to table R2, say t (const1, const2, const3). The following steps are performed sequentially:

- The expression signature list for data source R2 is searched. For each entry, the constant list or index for that expression signature is searched for entries that match the attribute values of the token t which are const1, const2 and const3 respectively. As per our example, the expression signature for R2 looks like R2.a = constant value. The list of constant values includes R2.a = val1, which has the trigger foo_bar linked to it.
- For every constant value that matches the given token t, the list of triggers for that constant value has to be checked. In our example, as foo_bar is a member of constant R2.a = val1, it needs to be checked for token t. The list of triggers for every constant

set under every expression signature has a pointer to both the rest of the predicate for the trigger and the alpha node id for this tuple variable.

- The values of token t are plugged into rest of the predicate and the rest of the predicate is checked to see if it evaluates to true.
- Now we have arrived at the final step of forming the select statement using the query templates of Treat node α2 for tuple variable R2. A token reaching the α2 means that it has already satisfied the selection predicate on tuple variable R2. The query template for alpha node α2 looks like:

  FROM clause: R1, R3

  TARGET clause: R1.* **,** all attribute values of an R2 tuple, R3.*

  WHEN clause: R1.b = R2.b and

  R1.a = val1 and

  R3.c = R2.c and

  R3.a = val3

  Now we plug in the values for the underlined variables R2.b and R2.c from token t, which are const2 and const3 respectively. The select statement thus formed using a technique similar to query processing by decomposition looks like:

  Select t.*, R1.*, R3.*

  From R1, R3

  Where R1.b = const2 and

  R1.a = val1 and

  R3.c = const3 and

  R3.a = val3

The trigger action is executed for every tuple selected by this select statement and is not fired if nothing is selected.

4.5 Conclusion

The technique used to process the join triggers during compile time when the trigger is created and during run time when the token is processed to see whether it fires any trigger looks similar to the decomposition technique used to process queries in Ingres. Especially the technique of plugging the values from the token into the select statement is an improvisation of the Ingres query processing technique using decomposition. The concept of query templates has been contrived to aid this technique of tuple substitution during run time. In the next chapter, we discuss the various issues that prompted us to choose the virtual Treat network over the Gator and the Rete network and why we decided to have virtual alpha nodes instead of materialized alpha nodes in our Treat network.

CHAPTER 5
IMPLEMENTATION ISSUES

5.1 Introduction

The previous chapter presented the TriggerMan join query processing technique and explained how it is somewhat similar to the decomposition-based query processing of Ingres. In this chapter we discuss why this technique was chosen, and its advantages and pitfalls compared to some available alternatives.

5.2 Why Decompostion-based Join Trigger Processing

As discussed in chapter 2, for single table triggers, TriggerMan outperforms conventional synchronous trigger processing database systems due to two main reasons. It is an asynchronous trigger processor and the entire process of selection predicate testing takes place in main memory. But join trigger processing in TriggerMan has to be carefully designed because there is no way the entire process can be completed in main memory without querying the database. So whichever technique we choose to implement join trigger processing, it has to conform to some basic requirements like reasonable performance and correctness apart from not having any negative effects on the performance of single table triggers. This prompted us to choose a technique that tested single table conditions first and then performs the joins only if necessary. This technique is effective, performs well and is easy to implement. To sum it up, the main reasons for

choosing the variation of the decomposition-based query processing technique of Ingres to implement join trigger processing are as follows:

- The technique had already been successfully implemented and found to perform well in Ingres.
- The variation of the decomposition-based query processing technique used in TriggerMan is not only straightforward to implement but also gives a reasonable performance.
- This technique does not impact the performance of single table triggers at all.
- The technique also takes advantage of the high quality query processor of the DBMS.

The rest of this chapter talks about a few other design decisions and then analyzes the results for various join trigger tests checking for performance and scalability.

5.3 Other Design Issues

Let us now discuss a few more critical decisions while designing the system and discuss the reasons why we chose them. One of the crucial decisions was to choose the discrimination network for join trigger processing in TriggerMan. We considered three choices to choose between namely, the Treat network, the Gator network and the Rete network.

A Rete network differs from a Treat network as it has a type of node called Beta node that stores the results of AND nodes. One Beta node is created for each AND node. Gator (Generalized Treat/Rete) networks [Hans93] contain the same type of nodes as that of Rete networks but with the difference that an AND node in a Gator network can join

any number of alpha or Beta nodes. Only A-Treat [Hans94] and Gator networks support both stored and virtual alpha nodes.

The reasons why the Treat network is chosen over the Rete and the Gator network are as follows:

- As TriggerMan is an asynchronous trigger processor, consistency of behavior is important, yet hard to achieve. If at all we choose to materialize the alpha nodes (whatever may be our discrimination network structure), it would have been difficult to maintain consistency in the information across the materialized nodes and the database. So we decided to have pure virtual alpha nodes. Once this decision was made, dismissing Rete network was easy, as the Beta nodes may get corrupted if they are under virtual alpha nodes.
- Gator (Generalized Treat/Rete) networks, when tuned properly usually outperform both Treat and Rete networks. But the future plan of parallel token processing may be severely hampered if Gator network is used as the discrimination network [Park99] in TriggerMan because parallel token processing causes problems in the semantic correctness of trigger processing. Park concludes that higher the consistency level, lower is the resulting performance [Park99]. This prompted the decision to stick with the virtual Treat network.

5.4 Performance Issues

As discussed in Chapter 2, triggers are a kind of alerters that fire when an event of interest occurs. This may be an insert/delete/update to an underlying database table. As Triggerman processes the update descriptors for these database operations

asynchronously, it scales well and thus allows thousands of triggers to be defined on a single database table. Due to the fact that a widely varying number of triggers can be defined and the cost to check different trigger conditions may also vary, it does not take the same time to process every update descriptor. Some of them may just fire a few triggers whereas some others can fire thousands of triggers. One of the main objectives of join trigger processing is not to lose the consistency in performance when the number of triggers grows. The performance of the system is expected to be relatively stable whatever may be the number of triggers defined on the system. At any time, with any number of triggers defined, the performance should not deteriorate.

5.5 Performance Analysis – Test 1

The first test (Test 1) for our performance analysis serves dual purposes. It

- checks for the system's scalability
- shows how the expression signature defined for each trigger selection predicate acts as a filter for the join triggers and how it helps us in avoiding an unnecessary query to the database.

Let us discuss this test in detail. We consider two database tables from a store database for this test, namely

Order1 (ono, categoryno, description) and

Order2 (ono, categoryno, description)

Part A of Test 1 - Scalability Test

The objective of the first part of Test 1 is to check for the scalability of TriggerMan. In order to accomplish our objective, we create a total of thousand join triggers on Order1 and Order2. They are all simple two-way join triggers as given below:

Create trigger [trigger_name]

From Order1, Order2

When Order1.ono = 1 and

Order1.categoryno = Order2.categoryno and

Order2.ono = 100

Do echo '[trigger_name] fired'

Let us name these triggers as trigger1, trigger,…, trigger1000.

The system takes a total of 132 seconds to create these 1000 triggers, which is effectively 7.5 triggers/sec. This is reasonably quicker when compared to the conventional database systems. The interesting aspect of this test is that a sizable amount of this total time is spent in writing to the trigger catalogs to facilitate the re-creation of these new triggers after boot time. So once the triggers are created, then every time we boot the system, it is going to take even less time to create these triggers as we need not write to the trigger catalogs now.

Part B of Test 1: To check the effectiveness of the TriggerMan Selection Predicate-Indexing scheme

The second part of Test 1 aims at showing the effectiveness of our Selection Predicate Indexing scheme [Hans98]. In order to achieve this objective, we do an insert into Order1 followed by 2 inserts into Order2. The first insert into Order1 does not fire

any trigger, as Order2 is empty. We design the next 2 inserts into Order2 in such a way that one insert to Order2 fires all the 1000 triggers and the other insert to Order2 does not satisfy the expression signature for Order2 and thus fails to fire any trigger. Let us look at each of the inserts into Order 1 and Order2 and discuss what happens.

Insert into Order1 values (1, 1, “jeans”) – T1

T1 does not fire any triggers even though it satisfies the expression signature on Order1 because the table Order2 is empty.

Insert into Order2 values (10, 1, “shirt”) – T2

T2 does not fire any triggers and takes 10 milliseconds. T2 checks for the expression signature on Order2 namely Order2.ono = constant. The constant list for Order2 has only one value namely 100 as all the 1000 triggers have the same Selection Predicate, Order2.ono = 100. As it does not match the expression signature condition, the remaining trigger conditions need not be checked. Thus our Selection Predicate-indexing scheme acts as a filter, effectively saving us time to run 1000 queries to the database. So the time taken for checking 1000 triggers in TriggerMan is a lot lesser than other trigger processing systems. Thus the Selection Predicate-indexing scheme of TriggerMan gives a tremendous performance boost.

Insert into Order2 values (100, 1, “pants”) – T3

T3 fires 1000 triggers in 4.857 seconds. As T3 satisfies the expression signature on Order2 (Order2.ono = 100), it forces the join trigger processing system to create a select statement using the query templates like

Select T3.*, Order1.*

From Order1

Where Order1.ono = 1 and

Order1.itemno = 1 (which is T3.itemno)

A query similar to this has to be issued for each of the 1000 triggers, which is the reason for the time taken.

5.6 Performance Analysis – Test 2

Test 2 illustrates how join trigger processing mechanism of TriggerMan avoids a query to a large table by using the selection predicate index to overcome it. We consider a three-way join trigger to illustrate this feature. We consider an additional table Order3 (ono, categoryno, description) for Test 2 as the rule has a three-way join featuring three tables. We take the following query for Test 2:

Create trigger trigger1001

From Order1, Order2, Order3

When Order1.ono = 1 and

Order1.categoryno = Order2.categoryno and

Order2.ono = 100 and

Order2.categoryno = Order3.categoryno and

Order3.ono = 1000

Do echo ‘trigger1001 fired’

Order1, which has 5000 rows is a large table compared to Order2 and Order3, which have only 50 rows. We do an insert into Order1 and calculate the time to complete

the trigger processing and an insert to the smaller table Order2 and calculate the time for trigger processing. The inserts are designed to fire the trigger. Let us discuss the inserts in detail.

Insert into Order1 (1, 1, ‘trousers’) – T1

Insert into Order2 (100, 1, ‘shorts’) – T2

T1 takes 70 milliseconds to fire trigger trigger1001 while T2 takes 100 milliseconds to fire the same trigger. The difference is due to the fact that T2 has to query a much larger table, Order1 and a relatively smaller table, Order3 to check for trigger conditions while T1 queries two smaller tables, namely Order2 and Order3.

5.7 Contribution to TriggerMan

My personal contribution to TriggerMan includes the implementation of join trigger processing, both at compile time to create the join triggers and at run time to process the tokens and check for join triggers. I have written approximately 2500 lines of code apart from designing the system under my professor’s guidance. The coding part covers some major modules like spi.c (which creates the expression signature for each trigger), token.c (which processes the update descriptors at run time), treat.c (which creates the treat network for join triggers) and rcg.c (which creates the rule condition graph for every trigger). My contribution to spi.c and rcg.c includes sizable modification to the data structures to handle join triggers. I modified token.c to process tokens that check for join triggers. Treat.c was entirely my work that includes the creating of the treat network for every trigger and also implementing the concept of query templates for every table.

CHAPTER 6
SUMMARY AND FUTURE RESEARCH DIRECTIONS

6.1 Summary

This thesis discussed the design and implementation of a decomposition-based solution to join trigger processing in an asynchronous trigger processor called TriggerMan. Initially, in the first few chapters we discussed the emergence of triggers in active databases. We also discussed the conventional synchronous trigger processing systems and their drawbacks. Chapter 2 talked about the asynchronous trigger processor, TriggerMan and also discussed its advantages over the synchronous trigger processing systems. The problem at hand, join trigger processing in TriggerMan and the motivation to solve it, namely decomposition-based query processing was discussed in chapter 3.

Chapters 4 and 5 discussed the implementation of join trigger processing in TriggerMan using the decomposition-based technique similar to the one used in Ingres and the reasons for choosing this technique. As TriggerMan performed exceptionally well for single table triggers, it imposed certain pre-conditions for join trigger processing. So whatever technique we use to process join triggers has to comply with these TriggerMan standards. It has to function correctly, perform reasonably well and in addition, should not adversely affect the performance of single table triggers. This prompted us to look at solutions that have already been implemented and tested so that they are both simple to implement and at the same time perform well in real time scenario. Decomposition-based

query processing from Ingres was one such technique that had our attention. The advantage of this approach with respect to correctness is that we never cache database data. So there is no possibility of cache corruption. As explained in this dissertation, a novel variation of this technique when used to implement join trigger processing, performed well in TriggerMan.

6.2 Future Research

One potential avenue for future research is replacing the Treat network with the more versatile, better performing Gator network as the discrimination network in TriggerMan. But this would require taking care of the consistency issues when we use the materialized alpha nodes in the Gator network. As TriggerMan is an asynchronous trigger processor, it will be a challenging project to research and implement a fully functional Gator network with a mixture of virtual alpha nodes (for highly selective indexed selection predicates), materialized alpha nodes (for less selective selection predicates) and beta nodes for storing the intermediate results. Even though condition testing using an optimized Gator network for multi-join triggers [Bodagala98] would definitely bring about a speedup when compared to condition testing using the Treat or the Rete network, taking care of the consistency issues is going to be tricky. Also if Gator network is used as the discrimination network for condition testing in TriggerMan, parallel token processing [Park99] may raise some more interesting issues.

The technique of caching the intermediate results of frequently occurring queries with a cache timeout is another interesting area of research. A set of frequently occurring queries maybe chosen from the set of existing triggers on first use. The results and the

associated queries would be stored in a table. Then whenever the system runs a query to check for trigger conditions, the cache table is scanned for a matching query. If there is one, the results are checked for their age and if it is less than the cache timeout, then the results of the query are taken from the table. This effectively saves us the time to run a SQL query. Now let us discuss the probable advantages and drawbacks of this technique. This technique will provide a performance boost only if the cache timeout is sizable because if it is too low, then the results in the cache table become stale soon, which may not give us the required performance enhancement. This technique may be suitable for triggers on infrequently updated tables because if the tables are updated frequently, then the cache timeout is going to be low.

A more interesting and effective technique may be a slight variation of the previous cache method. We could declare certain data source tables as “cacheable” along with a cache timeout while creating them. The term cacheable means that either the whole table (if and only if the table is small) or the alpha nodes on this table are cacheable. The term cache timeout refers to the frequency of updating either the materialized alpha nodes or the whole table whenever they are accessed depending on whether the data is stale or not. By stale date, we mean the data that is older than the cache time out. Let us now analyze the possible advantages and disadvantages of this idea with an example trigger. Let us assume we have triggers defined on 5 database tables namely R1 (a, b, c, d e), R2 (a, b, c, d, e), R3 (a, b, c, d, e), R4 (a, b, c, d, e) and R5 (a, b, c, d, e). Let R1, R2 be defined as cacheable with a cache time out of 1 hour.

```
Create Trigger foo_bar
From R1, R2, R3, R4, R5
When R1.a = val1 and
        R1.b = R2.b and
        R2.a = val2 and
        R2.c = R3.c and
        R3.a = val3 and
        R3.d = R4.d and
        R4.a = val4 and
        R4.e = R5.e and
        R5.a = val5 and
        R3.b = R1.b
do echo ”foo_bar fired”;
```

The RCG for foo_bar is shown in Figure 6.1.

As R1 and R2 have been defined as cacheable, once foo bar is defined, we materialize the alpha nodes on R1 and R2 for R1.a = val1 and R2.a = val2. Whenever there is an update/insert/delete to any of these tables, then the system automatically checks for foo_bar. Say we get an update to R3 called t (a1, b1, c1, d1, e1) and if it matches a1 = val3, then the next step as per our original join trigger processing will be issue a select statement like

```
Select R1.*, R2.*, t.*, R4.*, R5.*
From R1, R2, R4, R5
Where R1.a = val1 and
```

R1.b = R2.b and

R2.a = val2 and

R2.c = c1 and

R1.b = b1 and

a1 = val3 and

R4. = d1 and

R4.a = val4 and

R4.e = R5.e and

R5.a = val5

But by implementing the cache technique i.e., materializing the alpha nodes on R1 and R2, a call to the database can be avoided if there are no tuples matching any of these conditions: R1.a = val1 or R2.a = val2 or R2.c = c1. This can work wonderfully well and can provide a very effective speedup if the selection predicates on R1 and R2 are selective. But let us look at the disadvantages:

1. The trigger processing system has to have its own query processor to check for join conditions from the cached data. For example, to check for R2.c = c1, we need a query processor of our own because this is equivalent to running a select statement on a set of rows in main memory.
2. When the cached data on R1 or R2 are stale (older than the cache time out), then we need to update them before checking on them.
3. This may never guarantee consistency of data across the database table and the materialized alpha nodes.

On the other hand, we do have lots of advantages:

1. It may save us a call to the database table whenever the selection predicate on R1 and R2 are selective. So we can use the materialized alpha nodes as filters to prevent the access to the database.
2. Even if the selection predicates are satisfied, the modified query to the database contains a fewer number of tables than the original query.

Thus the caching technique may result in a tremendous speedup with the tradeoff being the complications incurred.

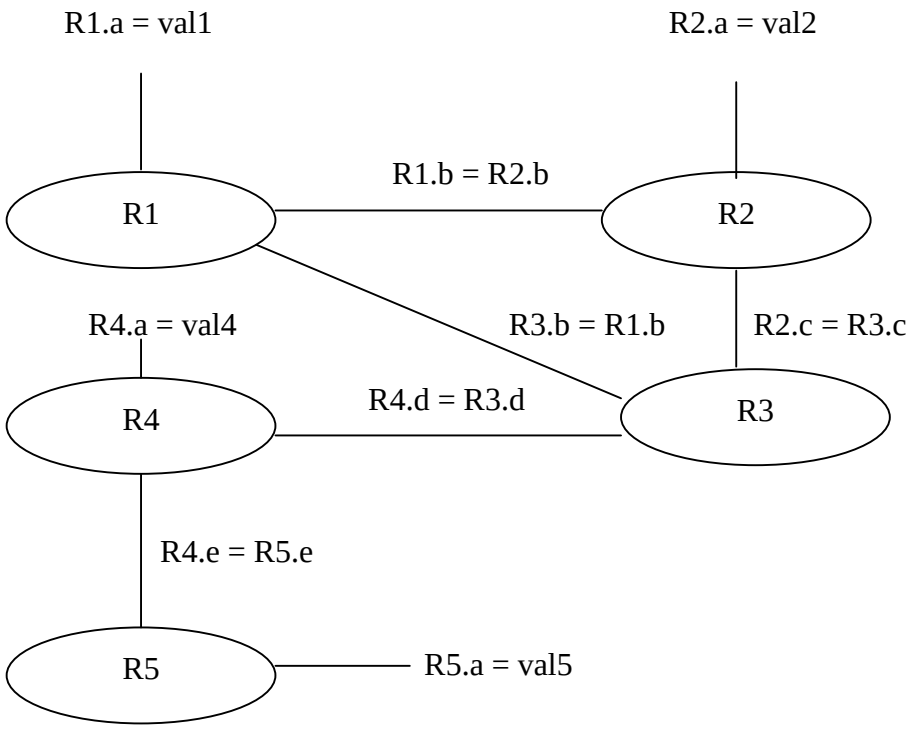

Figure 6.1: RCG for trigger foo_bar

Another important avenue of research in TriggerMan is thinking of ways to speed up trigger processing by incorporating parallelism in TriggerMan. There can be three ways of doing this. We can have token-level parallelism whereby multiple TriggerMan drivers' process the update descriptors in parallel. Then there is condition-level parallelism where we can check for trigger conditions in parallel. We can also think of action-level parallelism where we can execute the trigger actions in parallel.

A totally different avenue of research (that does not concentrate on performance improvement) is to extend the join trigger processing capabilities to User Defined Data Types, binary fields and blobs. The problem in extending the present join trigger processor to accommodate the UDTs, binary fields and blobs is that we need a textual representation for these types in the SQL statement we run to check for join trigger conditions. One probable solution to extend join trigger processing would be to have an internal textual representation for all these UDTs. Then, during run time, we would have to convert the external representation of the UDT to our textual format and then run our query to check for join trigger conditions. The solution to processing binary fields would be a two-way format conversion, one to convert binary fields to ascii text to run our SQL query and then the next one will be to convert the result back to something like hexa-decimal format to display the answers. The problem with Blobs can be handled by always using a handle for the blob object instead of just dealing with the blobs themselves. In this way, we may save time and space.

So an optimal and ideal TriggerMan trigger processing system may have Gator network as the discrimination network with all the consistency issues sorted out, have in-built caching mechanism with all the accessories like query processor to supplement the

trigger processing with caching along with the three levels of parallelism incorporated. Much work remains before a system such as this can be realized.

REFERENCES

[Alf98] Al-Fayoumi, N., Temporal trigger processing in the TriggerMan active DBMS. Ph.D. dissertation, University of Florida, Gainesville, 1998.

[Cha89] Chakravarthy, S., HiPAC: A research project in active, time-constrained database management. Final Technical Report. *Technical Report XAIT-89-02.* Xerox Advanced Information Technology, Cambridge, MA, August 1989.

[For82] Forgy, C. L., Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19(1):17-37, September 1982.

[Geh91] Gehani, N., & Jagadish, H. V., Ode as an active database: Constraints and triggers. In *Proceedings of the seventeenth International Conference on Very Large Data Bases 1991*, pages 327-336. Morgan Kaufmann, San Francisco, September 1991.

[Han90] Hanson, E. N., Chaabouni, M., Kim, C., & Wang, Y., A predicate matching algorithm for database rule systems. In *Proceedings SIGMOD Conference* 1990, pages 271-280. ACM Press, New York, June 1990.

[Han94] Hanson, E. N., & Chaabouni, M., The IBS tree: A data structure for finding all intervals that overlap a point. Technical Report WSU-CS-90-11. Wright State University, Dayton, OH, April 1990.

[Han96a] Hanson, E. N., The design and implementation of the Ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering* 8(1):157-172, February 1996.

[Han96b] Hanson, E. N., & Johnson, T., Selection predicate indexing in active databases using Interval Skip Lists. *Information System*s, pp. 269-298, 1996.

[Han97] Hanson, E. N., Bodagala, S., & Chadaga, U., Optimized trigger condition testing in Ariel using Gator Networks, University of Florida CISE Department. Technical Report 97-021, Gainesville, November 1997.

[Han98] Hanson, E. N., Konyala, M., Vernon, A., Noronha, L., & Park, J., Scalable trigger processing in TriggerMan. University of Florida CISE Department. Technical Report 98-008, Gainesville, July 1998.

[Han99] Hanson, E. N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parasarathy, S., Park, J., & Vernon, A., Scalable Trigger Processing. In *Proceedings International Conference on Data Engineering* 1999, pages 266-275. IEEE Computer Society, Los Alamitos, CA, March 1999.

[Kon98] Konyala, M., Predicate indexing in TriggerMan. Master's thesis, University of Florida, Gainesville, 1998.

[Mir87] Miranker, D. P., TREAT: A better match algorithm for AI production systems. In *Proceedings AAAI-87 Sixth National Conference on Artificial Intelligence*, pages 42-47. Morgan Kaufmann, San Francisco, August 1987.

[Par98] Park, J., Implementation and performance tuning of discrimination networks for asynchronous trigger processing view maintenance. Unpublished paper, University of Florida, Gainesville, 1998.

[Par99] Park, J., Parallel token processing in an asynchronous trigger system, Ph.D. dissertation, University of Florida, Gainesville, 1999.

[Pat98] Paton, N. W., ed. *Active rules in database systems.* Springer Verlag, New York, 1998.

[Rao98] Rao, J., & Ross, K. A., Reusing invariants: A new strategy for correlated queries. In *Proceedings SIGMOD Conference* 1998, pages 37-48. ACM Press, New York, June 1998.

[Sil98] Silberschatz, A., Korth, H. F., & Sudarshan, S., *Database management systems*, 3rd ed., McGraw-Hill, New York, 1998.

[Sis95a] Sistla, A. P., & Wolfson, O., Temporal conditions and integrity constraints in active database systems. In *Proceedings SIGMOD Conference 1995*, 24(2):269-280, San Jose, CA, June 1995.

[Sis95b] Sistla, A. P. & Wolfson, O., Temporal triggers in active databases. *IEEE Transactions on Knowledge and Data Engineer*ing 7(3):471-486, June 1995.

[Sto76] Stonebraker, M., Wong, E., Kreps, P., & Held, G., The design and implementation of INGRES. TODS 1(3): 189-222(1976).

[Sto91] Stonebraker, M., & Kernnitz, G., The POSTGRES next-generation database management system. *Communications of the ACM* 34(10):78-92, October 1991.

[Wan92] Wang, Y. W., & Hanson, E. N., A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proceedings IEEE Data Engineering Conference 1992*, pages 88-97. IEEE Computer Society, Los Alamitos, CA, February 1992.

[Wid95] Widom, J. & Ceri, S., eds., *Active database systems: Triggers and rules for advanced data processing*. Morgan Kaufmann, San Francisco, September 1995.

[Won76] Wong, E., & Youssefi, K., Decomposition – A strategy for query processing. *TODS* 1(3): 223-241(1976)

BIOGRAPHICAL SKETCH

Sasi Kumar Parthasarathy was born on June 5, 1975, in Jalarpettai, India. He received a bachelor's degree in computer science and engineering, securing first class with honors, from Regional Engineering College, Trichirapalli, in May 1997.

He joined the University of Florida in August 1997 to pursue a master's degree in computer and information science and engineering.

He has worked as a research assistant with Dr. Hanson at the Database Systems Research and Development Center.

His research interests are in active database systems.