Optimized Trigger Condition Testing in Ariel using Gator Networks*

Eric N. Hanson, Sreenath Bodagala, Ullas Chadaga,
Mohammed Hasan, Goutam Kulkarni and Jayashree Rangarajan

Rm. 301 CSE, P.O. Box 116120
CISE Department
University of Florida
Gainesville, FL 32611-6120
hanson@cise.ufl.edu
http://www.cise.ufl.edu/~hanson/

20 February 1997

TR 97-002

Abstract

This paper presents an active database discrimination network algorithm called Gator, and its implementation in a modified version of the Ariel active DBMS. Gator is a generalization of the widely known Rete and TREAT algorithms. Gator pattern matching is explained, and it is shown how a discrimination network can speed up condition testing for multi-table triggers. The structure of a Gator network optimizer is described. This optimizer can choose an efficient Gator network for testing the conditions of a set of triggers, given information about the structure of the triggers, database size, attribute cardinality, and update frequency distribution. The optimizer uses a randomized strategy to deal with the problem of a large search space. The results show that optimal Gator networks normally have a shape which neither pure Rete nor pure TREAT, but an intermediate form where one or a few inner joins (β nodes) are materialized. In addition, this study shows that it is indeed feasible to optimize Gator networks which perform rule condition testing more efficiently than either TREAT or Rete networks. In certain cases, the best Gator network is an order of magnitude or more faster than Rete and TREAT (a factor of 23 in one case).

1 Introduction

A crucial component of an active database system is the mechanism it uses to test trigger conditions as the database changes. This paper presents an efficient trigger (rule) condition-testing strategy based on a new type of discrimination network called the *Gator* network, or Generalized TREAT/Rete network [16, 3]. It is assumed here that a trigger condition can be based on multiple tables, and can involve both selections and joins, as in the Ariel active DBMS [4]. Gator networks are general structures that allow condition testing to be done for trigger conditions involving one or more tables. In general, there are many possible Gator networks for a given trigger condition, just as there are many possible query execution plans for evaluating a given query. Each will allow

*This work was supported by National Science Foundation grant IRI-9318607.

the trigger condition to be tested correctly, but some are much more efficient than others. Hence, an optimizer has also been developed for choosing a good Gator network for a trigger.

Rete and TREAT are rule condition testing structures that have been used both in production-rule systems such as OPS5, and in active database systems [2, 1, 4]. It has been observed in a simulation study that TREAT normally outperforms Rete, but the "right" Rete network can vastly outperform TREAT in some situations [21]. The reason that TREAT is usually better than Rete is that the cost of maintaining β nodes usually is greater than their benefit. However, if, for example, update frequency is skewed toward one or a few relations in the database, a particular Rete network structure can significantly outperform TREAT, as well as other Rete structures. It has been shown that Rete networks can be optimized, giving speedups of a factor of three or more in real forward-chaining rule system programs [13], which are like sets of triggers operating on a small, main-memory database. But even optimized Rete networks still have a fixed number of β nodes, which take time to maintain and use up space. With Gator, it is possible to get additional advantages from optimization, since β nodes are only materialized when they are beneficial.

This paper describes how trigger conditions can be tested using a Gator network, outlines a cost model for Gator networks, and presents how the Gator optimizer and trigger condition matching algorithm have been implemented in a modified version of Ariel [15]. Performance figures are given that demonstrate a substantial speedup in the trigger condition testing performance of Ariel.

2 Gator Networks

Gator networks are made up of the following general components:

selection nodes These test single-relation selection conditions against descriptions of database tuple updates, or "tokens." Selection nodes are also sometimes called "t-const" nodes, since they typically test tuples to see if they match constant values.

stored-α nodes These hold the set of tuples matching a single-table selection condition.

virtual-α nodes These are views containing a single-relation selection condition, but not the tuples matching the condition.

β nodes These hold sets of tuples resulting from the join of two or more α nodes.

P-nodes There is one P-node for each trigger. If a trigger only involves one table, then its P-node has a selection node as its input. If a trigger involves two or more tables, its P-node has as input one or more α and/or β nodes. If new tuples arrive at the P-node, the trigger is fired. The P-node is logically the root of a tree joining all the α and β nodes for the trigger.

root node The purpose of this node is to pass tokens to the selection nodes for testing. The root node is *not* the root of the join tree. The term "root" is used for historical reasons because it is used in the Rete algorithm [3].

By convention, the α nodes are drawn at the top, and the P-node is drawn at the bottom. In Gator networks for triggers involving more than one table, β nodes and P-nodes can have two or more child nodes, or "inputs." These inputs can be either α or β nodes. Every α and β node has a *parent* node that is either a β node or a P-node.

Rete and TREAT networks are special cases of Gator networks. Rete networks are always binary trees, with a full set of β nodes, all of which have two inputs. TREAT networks have no β nodes – all α nodes in a TREAT network feed into the P-node.

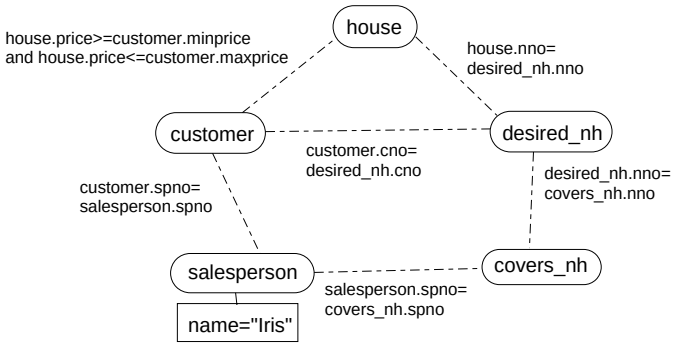


Figure 1: A rule condition graph for IrisRule.

To begin illustrating Gator networks with an example, consider the following schema describing real estate for sale in a city, real estate customers and salespeople, and neighborhoods in the city.

```
customer(cno,name,phone,minprice,maxprice,sp_no)
salesperson(spno,name)
neighborhood(nno, name, desc)
desired_nh(cno,nno)  ; desired neighborhoods for customers
covers_nh(spno,nno)  ; neighborhoods covered by salespeople
house(hno, spno, address, nno, price, desc)
```

A trigger defined on this schema might be "If a customer of salesperson Iris is interested in a house in a neighborhood that Iris represents, and there is a house available in the customer's desired price range in that neighborhood, make this information known to Iris." This could be expressed as follows in the Ariel rule language [4]:

```
define rule IrisRule
if salesperson.name = "Iris"
and customer.spno = salesperson.spno
and customer.cno = desired_nh.cno
and salesperson.spno = covers_nh.spno
and desired_nh.nno = covers_nh.nno
and house.nno = desired_nh.nno
and house.price >= customer.minprice
and house.price <= customer.maxprice
then raise event CustomerHouseMatch("Iris",customer.cno,house.hno)
```

The **raise event** command in the rule action is used to signal an application program, which would take appropriate action [7]. Internally, Ariel represents the condition of a rule as a *rule condition graph*, similar to a connection graph for a query [20]. The structure of the rule condition graph for IrisRule is shown in Figure 1. Sample Rete, TREAT and Gator networks for IrisRule are shown in figures 2, 3, and 4, respectively. Gator networks use objects called "+" tokens to represent inserted tuples, and "−" tokens to represent deleted tuples. Modified tuples are treated as deletes followed by inserts.

When a + token is generated due to inserting a tuple in a table, it is propagated through the Gator network to see if any triggers need to fire. Token propagation is explained below in

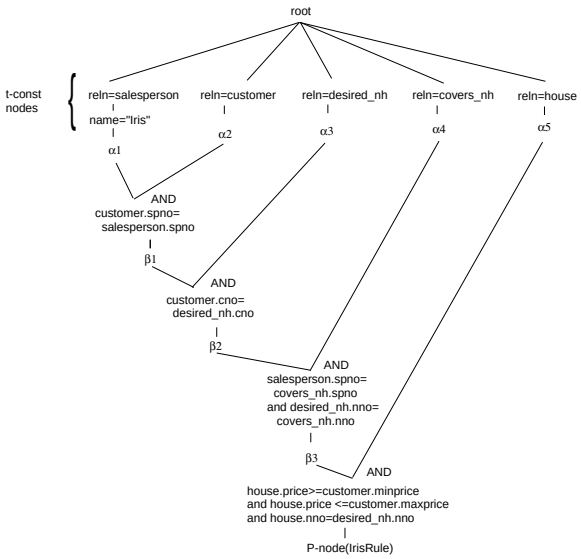


Figure 2: A Rete network for the rule IrisRule.

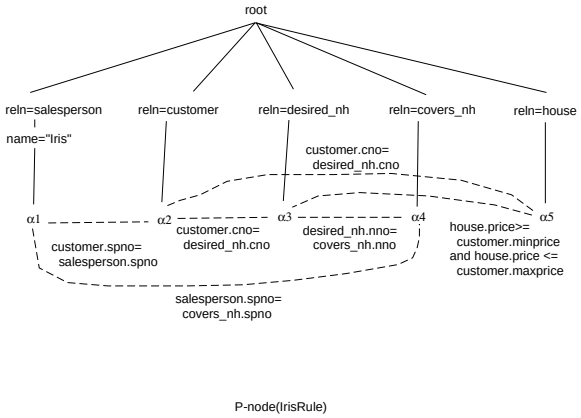


Figure 3: A TREAT network for the rule IrisRule.

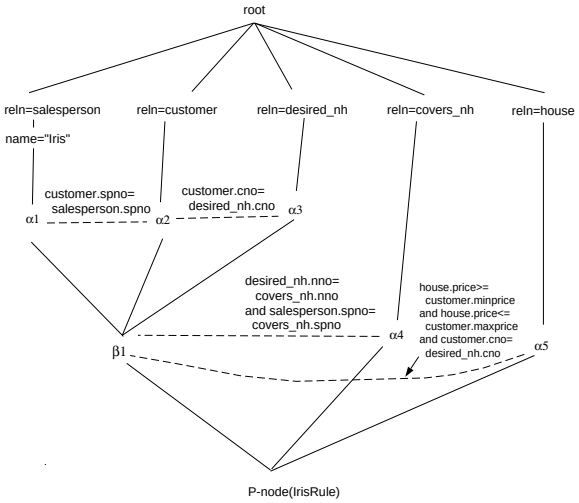


Figure 4: A Gator network for the rule IrisRule.

object-oriented terms, describing what happens when a token arrives at the types of nodes listed:[1]

root When the token arrives at the root node, the token is passed through a selection predicate index [6, 8] to reduce the set of selection nodes whose conditions must be tested against the token. The token is tested against each selection node that is not eliminated from consideration in the previous step. This identifies each α to which the token must be passed. The token is passed to each of these nodes in turn.

stored α node The tuple contained in the token is inserted into the node. The node will have a list of one or more other nodes called its "sibling" nodes, all of which have the same parent node. The token is joined with its siblings, using a specific join order that was saved at the time the Gator network was created (the choice of this join order is discussed in more detail later). A set of tuples is produced by this join operation. These tuples are packaged as + tokens and passed to the parent node.

virtual α node The work done is the same as that for a stored α node, except that the token is not inserted into the virtual α node.

P-node The rule is triggered for the data in the token.

β node The logic for this case is the same as for a stored α node.

As an example of Gator matching, suppose that the Gator network shown in Figure 4 is being used, and a new customer for Iris is inserted. This would cause the creation of a "+" token t_1 containing the new customer tuple. Token t_1 would arrive at $\alpha 2$ and be inserted into $\alpha 2$. Then, it could be joined with either $\alpha 1$ or $\alpha 3$. Assume that it is joined first with $\alpha 1$ where it matches with the tuple for Iris. The resulting joining pair is joined with $\alpha 3$. If elements of $\alpha 3$ join with this pair, each joining triple is packaged as a + token and forwarded to $\beta 1$. Upon arriving at $\beta 1$, a + token is stored in $\beta 1$. Then, the token can be joined to either $\alpha 4$ or $\alpha 5$ via the join conditions

[1] The actual Ariel implementation has a few other more specific types of nodes (see [4]), but the token propagation logic works as described here.

shown on the dashed edges from $\beta 1$ to $\alpha 4$ or $\alpha 5$, respectively. Assume it is joined to $\alpha 4$ first. The results would be joined next to $\alpha 5$. If a combination of tokens matched all the way across the three nodes $\beta 1$, $\alpha 4$ and $\alpha 5$ in this example, then that combination would be packaged as one + token and placed in the P-node, triggering the rule.

3 Cost Functions

As part of this work, cost functions were developed to estimate the cost of a Gator network relative to other Gator networks for a particular trigger. These functions are based on standard catalog statistics, such as relation cardinality and attribute cardinality, as well as on update frequency. The catalogs of Ariel have been extended to keep track of insert, delete and update frequency for each table. An update is considered equivalent to a delete followed by an insert, except in the special case of triggers that have ON UPDATE event specifications. The cost functions estimate the expense to propagate tokens through a Gator network, assuming a frequency of token arrival at different nodes determined by the frequency statistics, relation cardinality, attribute cardinality, selection and join predicate selectivity, and the presence of ON EVENT specifications for relations appearing in a trigger condition. In the analysis presented in this paper, insert, delete and update frequency are assumed to be equal (1/3 each). A presentation of the cost functions is beyond the scope of this paper. Details on the cost functions are presented elsewhere [5].

4 Optimization Strategy

For a given rule there can be many possible Gator networks. The efficiency of the rule condition testing mechanism depends on the shape of the Gator network used. An optimizer was implemented in Ariel that uses a randomized state-space search technique to get optimally shaped Gator networks. The use of a randomized approach to Gator network optimization was motivated by the fact that it has been used successfully for optimizing large join queries [10], a problem with a similarly large search space. Experiments were conducted [9, 17] which demonstrated that a randomized approach is superior to a dynamic programming approach like that used in traditional query optimizers [18].

Three randomized state-space search strategies were considered: iterative improvement (II), simulated annealing (SA) and two-phase optimization (2PO, a combination of II and SA). These generic algorithms require the specification of three problem-specific parameters, namely state space, neighbors function and cost function [12, 10, 11].

In the following discussion two sibling nodes in the discrimination network are said to be *connected* if the following holds. First, the *condition graph node set* of a Gator network node N, CGNS(N), is defined to be the set of condition graph nodes corresponding the the leaf α nodes of N. Two sibling Gator network nodes N1 and N2 are connected if there is a rule condition graph edge between an element of CGNS(N1) and CGNS(N2).

For the optimization of Gator networks, the following parameters were defined:

State Space The state space of the Gator network optimization problem for a given trigger is defined as the set of all possible shapes of the complete Gator network for that trigger. Each possible shape of the Gator network corresponds to a state in the state space. The state space is constrained so that no β node is created that requires a cross product to be formed among two or more of its children. It is assumed that all trigger condition graphs are connected, so

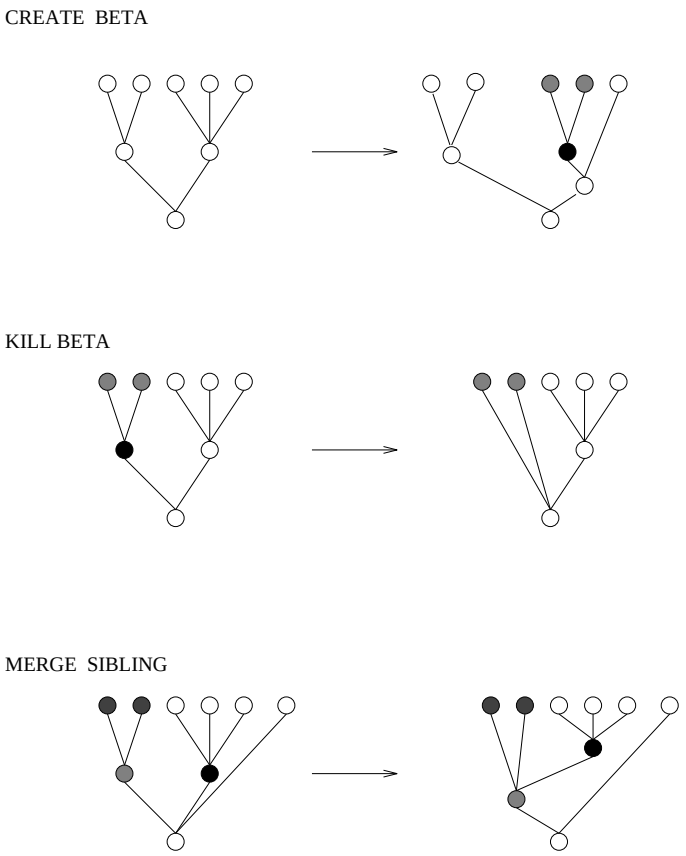


Figure 5: Local change operators.

it is always possible to find a Gator network that does not require cross products.[2]

Neighbors Function The neighbors function in the optimization problem is specified by the following set of transformation rules, which are also illustrated using examples in Figure 5.

- *Kill-Beta*: Kill-Beta removes a randomly selected β node, say KB, and adds the children of the node KB as children of the parent of the node KB.
- *Create-Beta*: Create-Beta adds a new β node, say CB, to the discrimination network. It first selects a random β node or the P-node (call this node PARENT). If PARENT has more than two children, Create-Beta randomly selects two connected siblings rooted at PARENT, makes them the children of CB, and makes CB the child of PARENT.
- *Merge-Sibling*: Merge-Sibling makes a node the child of one of its siblings. This operation first selects a random β node or the P-node. If this node has more than two children, then two connected siblings rooted at this node are randomly selected and one of them is made a child of the other. The node to which a child is added must be a β .

Cost Function The cost function is briefly outlined in section 3.

[2] If trigger condition graphs are not connected, the implementation adds dummy join edges with "true" as the join condition to make to make them connected.

The optimizer implemented is capable of using II, SA and 2PO. The 2PO strategy was used for all actual performance measurements. However, all three strategies are explained below since 2P0 is a combination of II and SA. Each of the II, SA and 2PO algorithms needs to be able to construct a random start state (feasible Gator network) given a condition graph for a trigger. Random start states are built in the following way:

1. Assume the condition graph has N nodes. Then N α nodes are created and inserted into a list.

2. While there is more than one element in the list, a number K where $2 \leq K \leq N$ is generated. A single starting element is selected from the list. Then, $K-1$ siblings for this node are selected from among the other elements of the list. This is done by following join edges leading out of the initially selected element to identify other elements of the list that have a join relationship with the initially selected element. The total of K elements identified are removed from the list, and a β node with them as children is formed. This β is inserted in the list.

When the list has only one element, that element is a complete Gator network for the trigger. A general description of II, SA and 2PO is given below.

4.1 Iterative Improvement

The Iterative Improvement (II) technique performs a sequence of local optimizations initiated at multiple random starting states. In each local optimization, it accepts random downhill movements until a local minimum is reached. This sequence of starting with a random state and performing local optimizations is repeated until a stopping condition is met. The final result is the local minimum with the lowest cost.

4.2 Simulated Annealing

Simulated Annealing (SA) is a Monte Carlo optimization technique proposed by Kirkpatrick et al. [14] for problems with many degrees freedom. It is a probabilistic hill-climbing approach where both uphill and downhill moves are accepted. A downhill move (i.e. a move to a lower-cost state) is always accepted. The probability with which uphill moves are accepted is controlled by a parameter called temperature. The higher the value of temperature, the higher the probability of an uphill move. However, as the temperature is decreasing with time, the chances of an uphill move tend to zero [14, 12].

4.3 Two Phase Optimization

In its first phase, 2PO runs II for a small period of time, performing a few local optimizations. The output of the first phase, i.e. the best local minimum, is input as the initial state to SA, which is run with a very low initial temperature. Intuitively this approach picks a local minimum and then searches the space around it. It is interesting to observe that this approach is capable of extricating itself out of the local minimums. However, the low initial temperature makes climbing very high hills virtually impossible. It has been observed that 2PO performs better than both II and SA approaches for optimizing large join queries [10]. The details of the actual implementation of 2P0 discussed in this paper, such as the crossover point between II and SA, the performance of II and SA individually, etc., are beyond the scope of this paper.

5 Modifications to Ariel

The first implementation of the Ariel active DBMS was based on the A-TREAT algorithm, which did not use β nodes. Ariel was thus modified to support β nodes. A discrimination network must be "primed," at the time a trigger is created; in other words, its stored α and β nodes must be loaded with data. Ariel's priming mechanism was modified to allow β nodes to be primed. Also, Ariel's token propagation strategy was modified to make use of and maintain β nodes.

5.1 New Discrimination Network Node Types

In the original Ariel system, there were seven different types of α nodes with slightly different behavior [4]. Memory nodes in Ariel can be either *static*, in which case their contents are persistent and are stored between transactions, or *dynamic*, in which case they are flushed after each transaction.

To implement Gator, the memory node class hierarchy was modified to include the following types of β nodes:

- **BetaMemory** This is the superclass of the other β node types.
- **StaticBeta** An ordinary β node. If none of the children of a β node is a dynamic node, i.e. neither dynamic-α or dynamic-β, then that β node is a *StaticBeta*.
- **DynamicBeta** If any of the children of a β node is a dynamic node, i.e. either dynamic-α or dynamic-β, then that β node is a *DynamicBeta*.
- **TransBeta** (short for Transparent Beta). An instance of this class is used at the root of the Gator network as a place holder for the P-node.

Virtual β nodes similar to virtual α nodes are not needed since the non-existence of a β node implies the need to reconstruct its contents as required.

5.2 Details on Priming

In Ariel, stored α and β nodes are primed. However, since the contents of dynamic α and β nodes do not outlive a transaction, they need not be primed. Also, the virtual-αs are not materialized during priming.

To prime a stored-α, a one-tuple-variable query is formed internally to retrieve the data to be stored in the α node. This one-variable query is passed to the query optimizer, and the resulting plan is executed. The data retrieved are stored in the α memory. To prime a β node, first its children are primed, and then the children are joined to find the data to put in the β node. The priming strategy used can be described with the following recursive algorithm. The Prime method is invoked on the root (P-node) of the Gator network to prime the network:

```
Prime(Node)
{
    if (childrenExist(Node))
        for each C in ChildOf(Node)
            Prime(C)
    materializeTuples(Node)
}
```

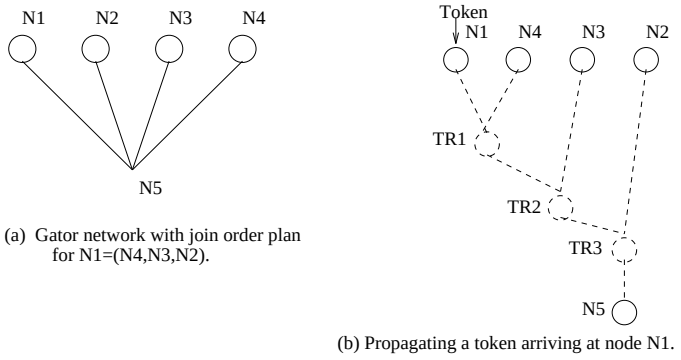


Figure 6: Join order plan for a Gator node.

The materializeTuples method forms the tuples for Node by running the one-variable query in the case of an α node, and by joining the children of Node in the case of a β node.

5.3 Generating Token Join Order Plans

Every node with a sibling in the Gator network has a join plan attached to it. The join plan is a sequence of two-way joins regulating the order in which tokens arriving at the node would be joined with each of its siblings. For instance, in Figure 6(a) the join plan attached to node N1 is (N4,N3,N2). When a token arrives at node N1, it is first joined with the contents of node N4. The resulting Temporary Result (TR) of the join is then joined with contents of the node N3 and so on, as shown in Figure 6(b). The TR's are not stored. They are generated dynamically and discarded.

An important objective is to choose a join plan with the minimum cost. However, since choosing token join plans must be done very frequently (hundreds or thousands of times) while finding an optimized Gator network for one trigger, it is too expensive to use traditional query optimization [18] to find the join order plan. Instead, the following heuristic is used: during each of the two-way joins, the current result should be joined with that sibling that would give the join result with smallest estimated size. This gives a reasonable join order plan quickly.

6 Performance Evaluation

This section presents the details of various experiments conducted to study the performance of Gator, Rete and TREAT discrimination networks. The performance metric in all the experiments is the rule condition evaluation time. The rule condition evaluation time is the time to evaluate a rule condition using a discrimination network (i.e. the time to pass a set of tokens through the network).

The Ariel active relational DBMS was used as a testbed for conducting all the experiments. The average rule activation time was measured by processing a randomly generated stream of updates. The table to which an update was applied was determined using a frequency distribution equivalent to the update frequency statistics maintained in the system catalog. Inserts were done on each table, and the time for each was measured. Then, a "total rule condition testing time" was calculated by multiplying the time spent propagating a token for each table by the insert frequency for the table.

Rules were created on a synthetically generated database and each of the three different discrimination networks, Rete, Gator and TREAT, were generated for them. The synthetic database

generated had the following properties. The relation sizes (number of tuples) were randomly chosen to be in the range [200, 300] or [2000, 10000]. For each relation, a range was randomly selected, and within each range, a number was randomly chosen that would be that relation's cardinality. The number of unique values of an attribute of a relation was chosen to give a roughly even mix of low and high cardinality attributes. The cardinalities were chosen as follows. For each attribute, a number s was chosen from one of two ranges, selected at random: [0.01, 0.1] or [0.5, 1.0]. The cardinality of the attribute was given by multiplying that relation's cardinality by s.

Experiments were performed on rules having the following types of Rule Condition Graphs (RCGs):

string type Each relation in the rule condition participates in a join with two other relations such that the rule condition graph looks like a string. The two relations at the two ends of the string participate in only one join. The following is an example of a rule with a string type RCG. R1, R2, R3 and R4 are the relations, `a` is an attribute of R1, `b` and `c` are attributes of R2, `d` is an attribute of R3 and `e` is an attribute of R4.

```
define rule Rule1
if R1.a = R2.b and R2.c = R3.d and R3.d = R4.e
then <action1>
```

star type One relation participates in a join with all the other relations in the rule condition. The following is an example of a rule with a star type RCG.

```
define rule Rule2
if R1.a = R2.b
and R1.c = R3.c
and R1.b = R4.d
then <action2>
```

The update frequency distribution of various relations in the database significantly affects the performance of discrimination networks. The following three update frequency distributions were chosen:

Skewed One of the relations has a very high update frequency and the other relations have low frequencies. In all cases, one relation (always R3) is assigned 0.8 as its update frequency, and the rest are assigned 0.05.

Even All the relations have the same update frequencies.

Ramp The frequencies of relations decrease in a ramp-like or stair-like manner. For all the cases, the following distribution was used: (R2=.4, R4=.3, R3=.2, R1=.05, R5=.05).

In all cases, frequencies sum to one.

The results of various experiments conducted on rules with five tuple variables, with different frequencies and with different rule condition graphs, are explained next. Two graph structures were considered. The first was a string type RCG with selection conditions on two of the five relations. The second was a star type RCG with selections on one of the five relations.

For all the tests discussed, an optimized Gator network is compared with a TREAT network (for which there is only one choice) and a non-optimized, arbitrarily chosen Rete network. Rete

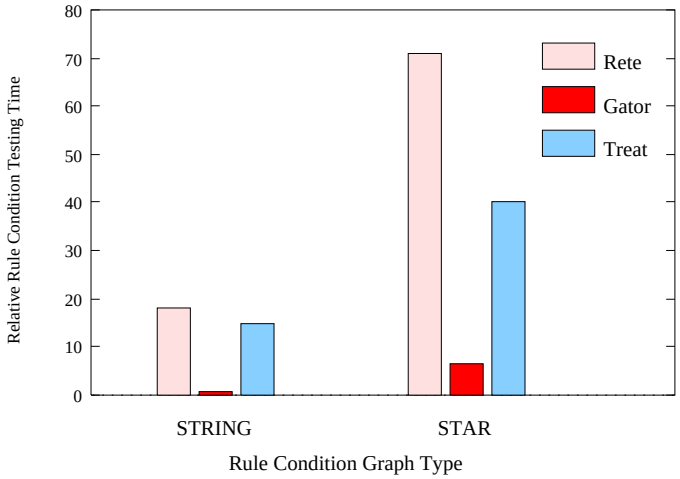


Figure 7: String and star type RCG and skewed frequency distribution.

networks can be optimized, but this was not done since the focus of this work is on the more general Gator structure. Figure 7 shows the results for a rule with five tuple variables, with string and star type RCGs and skewed frequency distribution.

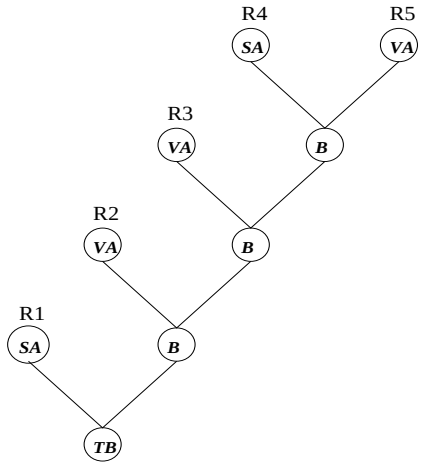
It can be seen that Gator is doing much better than Rete and TREAT (a factor of 23) in this case. The Gator, Rete and TREAT networks generated are shown in Figure 8. In all the networks, the optimizer decided to create virtual α nodes for relations with no selection predicate in the rule condition, preventing the duplication of relations and thus saving space. In Gator, the relation with high update frequency (R3=0.8) was pushed down the discrimination network toward the P-node. This means fewer token joins need to be done as tokens propagate through the network due to updates. Also, the stored α nodes with low size are at the top of the network which helps to reduce the size of the β nodes below them.

In the case of TREAT, shown in Figure 8, whenever a new token enters the network it always has to participate in a join with four other α nodes and that explains its higher rule condition testing time. In the case of Rete (A) in Figure 8, the virtual α node corresponding to the relation with the highest frequency (R3) is near the top of the network, and that means higher join processing costs and β-node maintenance costs.

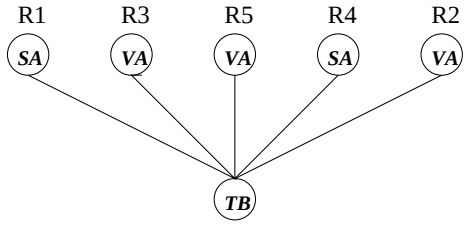
Figure 9 shows the results for the same rule with even frequency distribution. Gator again performs better than Rete and TREAT. The Gator network for this case is given in Figure 9 (D). The Gator network has a few β nodes. These result shows that it can be beneficial to maintain a few β nodes (though not as many as contained by a Rete) to get optimal performance. The intuition is that having a few β nodes in the right place reduces the number of high-cost joins to be performed, and the benefit of this is greater than the cost of maintaining the β nodes.

It was observed during these experiments that the speedup of Gator over Rete and TREAT for star type RCGs was less than for string type RCGs. It appears that the optimizer as implemented does not explore the search space in a smooth manner for star type RCGs. The reason for this seems to be that the connectivity between different relations is poor for star type RCGs. Addition of more edges to the rule condition graph with "true" as the join condition is being investigated. This will generate many more states in the state space, many of which will have a high cost, but it may allow a smoother transition between states that are "interesting" for the optimizer to explore.

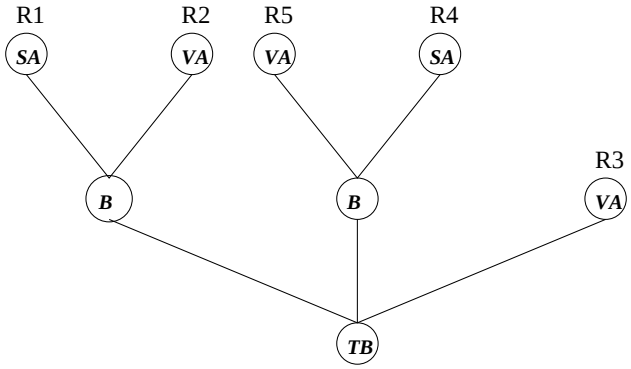
In many cases it is not intuitive why one network is better than another, in large part because there are so many competing factors that influence the performance of a network. Hence a con-



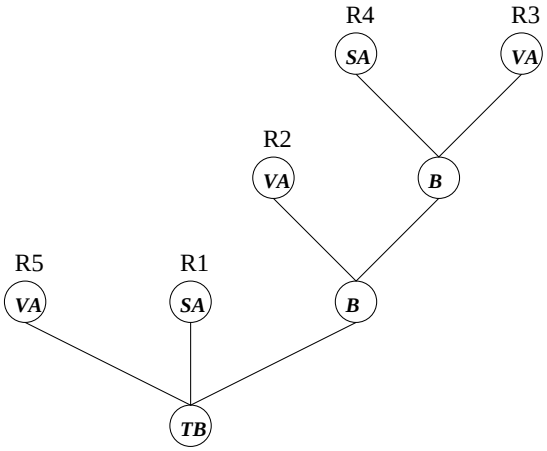
(A) Rete, Skewed



(B) Treat, Skewed



(C) Gator, Skewed



(D) Gator, even

Figure 8: Gator, Rete and TREAT networks generated for string type RCG. In this figure, SA=stored α, VA=virtual α, B=β, and TB=trans-β (P-node). The symbols R1 through R5 represent the base relations from which the labeled α nodes are derived.

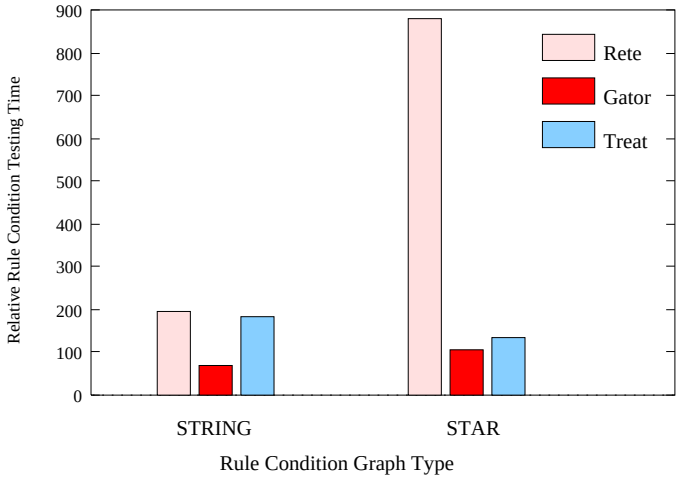


Figure 9: String and star type RCG and even frequency distribution.

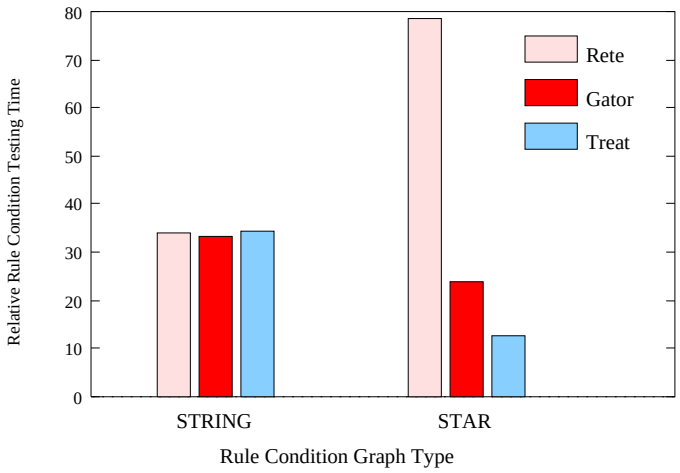


Figure 10: String and star type RCG and ramp frequency distribution.

clusion of this work is that it is better to use cost functions and search to perform optimization of Gator networks than to use heuristics to pick a good network.

Figure 10 shows the results for the string and star RCGs with a ramp frequency distribution. For the rule with the string RCG, the performance of the Gator network was slightly better than Rete and TREAT. However, for the rule with the star RCG, TREAT does better than Gator. It appears that the cost formulas may have an inaccuracy in this particular case which prevents the actual best Gator network from being found. However, overall, the results presented indicate that the cost formulas are accurate enough to allow a good Gator network to be found in a large majority of cases. Refinements to the cost formulas to better handle star type rule condition graphs are being considered.

7 Conclusion

This paper has introduced Gator networks, a new discrimination network structure for optimized rule condition testing in active databases. A cost model for Gator has been developed, which

is based on traditional database catalog statistics, plus additional information regarding update frequency. A randomized Gator network optimizer has been implemented and tested as part of the Ariel active DBMS.

An interesting result of this work is that for most cases, even for even update frequency distributions, the optimal Gator network has a few β nodes – it is not a TREAT network. In addition, this work shows that it is beneficial to use a general discrimination network structure (Gator), instead of limiting the possibilities to TREAT or Rete. Also, it shows that update frequency distribution has a tremendous influence on the choice of the best discrimination network. Moreover, it is indeed feasible to develop a cost model and search strategies that allow effective Gator network optimization.

This work has clearly demonstrated the value of optimizing the testing of trigger conditions involving joins in active databases. This can help make it possible to implement the capability to efficiently and incrementally process triggers with joins in their conditions in commercial database systems, thus making a new, powerful tool available to database application developers.

References

[1] David A. Brant and Daniel P. Miranker. Index support for rule activation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 42–48, May 1993.

[2] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming.* Addison Wesley, 1985.

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

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

[5] Eric N. Hanson, Sreenath Bodagala, Mohammed Hasan, Goutam Kulkarni, and Jayashree Rangarajan. Optimized rule condition testing in ariel using gator networks. Technical Report TR-95-027, University of Florida CIS Dept., October 1995. http://www.cis.ufl.edu/cis/tech-reports/.

[6] Eric N. Hanson, Moez Chaabouni, Chang-ho Kim, and Yu-wang Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 271–280, May 1990.

[7] Eric N. Hanson, I-Cheng Chen, Roxana Dastur, Kurt Engel, Vijay Ramaswamy, Chun Xu, and Wendy Tan. Flexible and recoverable interaction between applications and active databases. *VLDB Journal*, 1997. Accepted.

[8] Eric N. Hanson and Theodore Johnson. Selection predicate indexing for active databases using interval skip lists. *Information Systems*, 21(3):269–298, 1996.

[9] Mohammed Hasan. Optimization of discrimination networks for active databases. Master's thesis, University of Florida, CIS Department, November 1993.

[10] Yiannis Ioannidis and Younkyung Cha Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 312–321, May 1990.

[11] Yiannis Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 168–177, May 1991.

[12] Yiannis Ioannidis and Eugene Wong. Query optimization by simulated annealing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987.

[13] Toru Ishida. An optimization algorithm for production systems. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):549–558, August 1994.

[14] S. Kirkpatrick, C. C. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[15] Goutam Kulkarni. Extending the Ariel active DBMS with Gator, an optimized discrimination network for rule condition testing. Technical Report TR95-006, University of Florida, CIS Dept., February 1995. MS thesis, http://www.cis.ufl.edu/cis/tech-reports/.

[16] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proc. AAAI National Conference on Artificial Intelligence*, pages 42–47, August 1987.

[17] Jayashree Rangarajan. A randomized optimizer for rule condition testing in active databases. Master's thesis, University of Florida, CIS Department, December 1993.

[18] P. Selinger et al. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1979. (reprinted in [19]).

[19] Michael Stonebraker, editor. *Readings in Database Systems.* Morgan Kaufmann, 1994.

[20] Jeffrey D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[21] Yu-wang Wang and Eric N. Hanson. A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proc. IEEE Data Eng. Conf.*, pages 88–97, February 1992.