Optimized Trigger Condition Testing in Ariel Using Gator Networks*

Eric N. Hanson
CISE Department
University of Florida
hanson@cise.ufl.edu

Sreenath Bodagala
CISE Department
University of Florida
sre@cise.ufl.edu

Ullas Chadaga
CISE Department
University of Florida
uchadaga@cise.ufl.edu

12 November 1997

TR-97-021

Abstract

This paper presents an active database discrimination network algorithm called Gator. 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, along with a discussion of optimizer performance, output quality, and accuracy. 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. Optimizer validation was performed, showing a strong correlation between predicted cost of a Gator network and its actual cost when used for trigger condition testing. The results show that optimized Gator networks normally have a shape which is neither pure Rete nor pure TREAT, but an intermediate form where some but not all possible inner joins (β nodes) are materialized. In certain cases, the best Gator network is an order of magnitude or more faster than both optimized Rete and TREAT. Gator networks can also be used to optimize maintenance of materialized views.

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 [18, 6]. It is assumed here that a trigger condition can be based on multiple tables, and can be a non-procedural expression involving selections and joins, as in the Ariel active DBMS [7]. 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 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.

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

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 [4, 3, 7]. It has been observed in a simulation study that TREAT can outperform Rete, but the "right" Rete network can vastly outperform TREAT in some situations [25]. The reason that TREAT is sometimes better than Rete, particularly in a limited-buffer-space environment, is that the cost of maintaining materialized join (β) nodes sometimes 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 [15], 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. Performance figures are given that demonstrate a substantial speedup in trigger condition testing performance using Gator compared with optimized Rete and TREAT. The performance (running time) characteristics of the optimizer are given, along with a discussion of how the quality of the optimization results varies with different optimizer parameters. In addition, information on optimizer validation is included showing that the cost estimates the optimizer generates for Gator networks correlate well with the actual cost of using the networks for trigger condition testing.

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 and/or β 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 two 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. It is straightforward to use Gator networks for materialized view maintenance by replacing the P-node of the network with a β node designated as the materialized view.

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 [6].

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.

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 [7]:

```
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 [9]. Internally, Ariel represents the condition of a rule as a *rule condition graph*, similar to a query graph. The structure of the rule condition graph for IrisRule is shown in Figure 1 (A).

Sample Rete, TREAT and Gator networks for IrisRule are shown in figures 1 (B), 2 (A), and 2 (B), respectively.

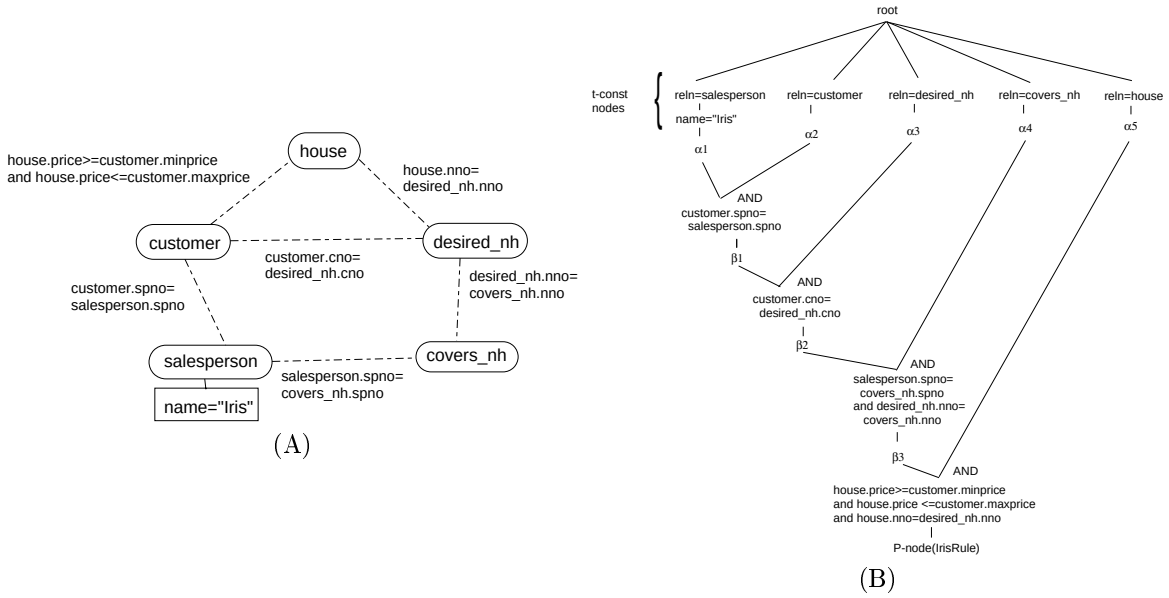


Figure 1: Rule condition graph (A) and Rete network (B) for IrisRule.

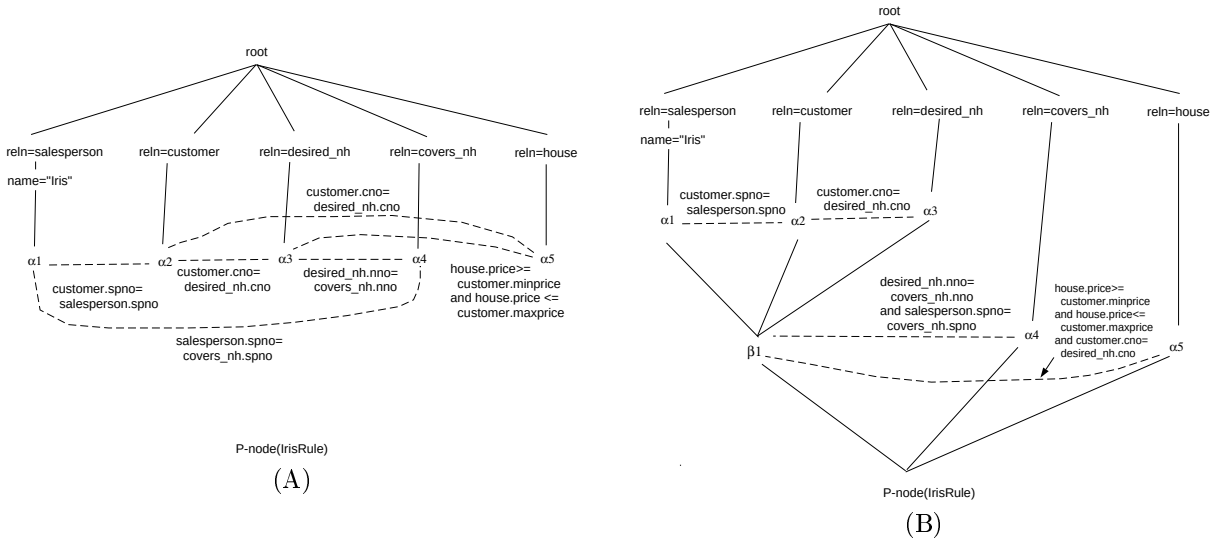


Figure 2: TREAT network (A) and Gator network (B) for IrisRule.

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. Propagation of + tokens is explained below in object-oriented terms, describing what happens when a token arrives at the types of nodes listed (− tokens are treated in a comparable fashion – details are omitted):[1]

root When the token arrives at the root node, the token is passed through a selection predicate index [8, 10] 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.

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

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

As an example of Gator matching, suppose that the Gator network shown in Figure 2 (B) 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 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.

[1] The actual Ariel implementation has a few other more specific types of nodes (see [7]), but the token propagation logic works as described here.

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.

The parameters relevant to the cost of a Gator Network are as follows:

CPU_{weight}:	The relative cost of a CPU operation
I/O_{weight}:	The relative cost of an I/O operation
$R(\alpha)$:	The base relation of the α-node, α.
N:	Any node in a discrimination network: α, β or a P-node.
$Sel(\alpha)$:	The selectivity factor of the selection predicate associated with an α-memory node, α.
$F_i(R)$:	The insert frequency of relation R relative to other relations.
$F_d(R)$:	The delete frequency of relation R relative to other relations.
$Pages(N)$:	The number of pages occupied by a node N.
$Card(N)$:	Cardinality of N.
$Card(N.attr)$:	Cardinality of attribute $attr$ in N.
$Card(R)$:	Cardinality of relation R.
$Card(R.attr)$:	Cardinality of attribute $attr$ in R.
$fanout$:	Fanout of a node in a B^+-tree. Cost functions involving indexes are given only for B^+-trees.

The cost model given below assumes that buffer space is limited, charging an amount I/O_{weight} for each I/O. However, the effect of having large buffer space can be approximated by setting I/O_{weight} to zero or some very small value. Both limited- and plentiful-buffer-space environments are addressed later in the paper. The small and large buffer space cost models are referred to as CM1 and CM2, respectively.

The cost formulas for a Gator network are defined recursively. The base case is for α nodes. The cardinality and insert and delete frequency for alpha nodes are as follows:

Cardinality, $\text{Card}(\alpha) = \text{Card}(\text{R}(\alpha)) \times \text{Sel}(\alpha)$
Insert Frequency, $F_i(\alpha) = F_i(R(\alpha)) \times Sel(\alpha)$
Delete Frequency, $F_d(\alpha) = F_d(R(\alpha)) \times Sel(\alpha)$

The cost of an α node is given by the cost of maintaining tuples that are stored in it. The insert cost of an α node, $C_i(\alpha)$, is the cost of inserting a tuple into it. The delete cost, $C_d(\alpha)$, is the cost of deleting a tuple from the α node. The insert and delete costs for virtual α nodes, stored α nodes with no indexes, and stored α nodes with an index on a column involved in a join with another memory node are as follows:

Insert Cost, $C_i(\alpha) = 0$, Virtual alpha node
$= CPU_{weight} + 2 \times I/O_{weight}$,
Stored α node with no index
$= CPU_{weight} + 2 \times I/O_{weight} + \lceil \log_{fanout} Card(\alpha) \rceil \times CPU_{weight}$,
Stored alpha node with an index on a join attribute

Delete cost, $C_d(\alpha) = 0$, Virtual alpha node
$= CPU_{weight} \times Card(\alpha) + I/O_{weight} \times (Page(\alpha) + 1)$,
Stored alpha node with no index
$= CPU_{weight} \times Card(\alpha) + I/O_{weight} \times (Page(\alpha) + 1) +$
$\lceil \log_{fanout} Card(\alpha) \rceil \times CPU_{weight}$,
Stored alpha node with an index on a join attribute

The complete formula for the cost of an α node is:

$$Cost(\alpha) = F_i(\alpha) \times C_i(\alpha) + F_d(\alpha) \times C_d(\alpha)$$

The cost of a β node is defined as the cost of each of its children, plus the local cost of joining tokens that arrive at each child with the other children, and of updating the stored β node itself.

Let $\prod_S (\beta) = \prod_{\alpha \in leaves(\beta)} Card(R(\alpha))$,

$\prod_\sigma (\beta) = \prod_{\alpha \in leaves(\beta)} Sel(\alpha)$ and

$\prod_\psi (\beta) = \prod (JSF(\alpha_i, \alpha_j))$, where α_i, $\alpha_j \in leaves(\beta)$ and $\exists edge(Rel(\alpha_i, Rel(\alpha_j))$ in the rule condition graph.

Define the cardinality of a β node, $\text{Card}(\beta) = \prod_\sigma (\beta) \times \prod_\psi (\beta) \times \prod_S (\beta)$

Let $TokenGenCount(N, \beta)$ represent the number of tokens generated at a β-node, β, due to a single token arriving at a child node, N, of β.

Define the insert and delete frequencies (F_i and F_d) of a β as:

$$F_i(\beta) = \sum_{N \in children(\beta)} F_i(N) \times TokenGenCount(N, \beta)$$

$$F_d(\beta) = \sum_{N \in children(\beta)} F_d(N) \times TokenGenCount(N, \beta)$$

$JoinSizeAndCost(N, \beta)$ estimates the cost of performing a sequence of two-way joins when a token arrives at node N, as well as the expected size of the result. The cost and size are calculated together for convenience since the calculations are done in a similar way. $JoinSizeAndCost(N, \beta)$ will be described in detail later. The value of $TokenGenCount(N, \beta)$ is obtained as follows:

$TokenGenCount(N, \beta)$ { (size, cost) = $JoinSizeAndCost(N, \beta)$; return(size) }

The cost of a β node is the sum of the following: (1) the cost of the children of the β node, (2) the

cost of performing joins for tokens fed into all the children of the β node, and (3) the cost associated with maintaining (updating) the β node. A formula for the cost of a β node is thus:

$$Cost(\beta) = LocalCost(\beta) + \sum_{N \in children(\beta)} Cost(N)$$

where

$$LocalCost(\beta) = \sum_{N \in children(\beta)} \{F_i(N) \times PerChildInsCost(N, \beta)\} + \{F_d(N) \times PerChildDelCost(N, \beta)\}$$

$PerChildInsCost(N, \beta)$ and $PerChildDelCost(N, \beta)$ indicate the respective costs of processing a + and − token arriving at a child N of the β node. $PerChildInsCost(N, \beta)$ consists of the cost a multi-way join performed on a tuple arriving at a child N of the β node and the cost of doing any needed inserts into the β node. Following the join order plan associated with the node N, a sequence of two-way joins with each of the siblings of this node is performed. $PerChildDelCost(N, \beta)$ is analogous to $PerChildInsCost(N, \beta)$. $JoinSizeAndCost(N, \beta)$ returns the estimated join size and the estimated join processing cost.

$PerChildInsCost(N, \beta)$ {
    (size, cost) = $JoinSizeAndCost(N, \beta)$
    insertCost = $\lceil \frac{size}{tuplesPerPage(\beta)} \rceil \times 2 \times I/O_{weight}$ + size $\times CPU_{weight}$
    return (cost + insertCost)
}

$PerChildDelCost(N, \beta)$ {
    (size, cost) = $JoinSizeAndCost(N, \beta)$
    deleteCost = (Yao(Card(β),Pages(β),TRSize) + Pages(β)) $\times$ I/O_{weight} + $Card(\beta) \times CPU_{weight}$
    return (cost + deleteCost)
}

Yao(n,m,k) function estimates the number of pages that would be touched when k tuples are randomly selected within relations/nodes that occupy m pages, each page containing n/m records.

Consider again $JoinSizeAndCost(N, \beta)$. In addition to size and update frequency information, its calculation is also based in part on the join order plan of the node N. Let TR represent the temporary join result formed during the join process, $TRSize$ represent the cardinality of TR and $TR.joinAttr$ represent the estimated cardinality of the join attribute in TR. JoinSizeAndCost(N,β) can be computed as follows:

$JoinSizeAndCost(N, \beta)$ {
    TRSize = 1
    TempCost = 0
    for each node n in the Join Order plan of N
    {
        TempCost = TempCost + $TwoWayJoinCost(TRSize, n)$
        TRSize = TRSize $\times$ Card(n) / max(Card(TR.joinAttr), Card(n.joinAttr))

```
    }
     return (TRSize, TempCost)
}
```

$TwoWayJoinCost(TRSize, n)$ represents the cost of performing a join between the Temporary Result (TR) of size $TRSize$ and a node n. There are two cases to consider for computing $TwoWayJoinCost$. The first case is when n is a stored α node (without an index on the join attribute), a β node, or a virtual α node without an index on the join attribute on its base relation (for this case, $Pages(n)$ refers to the pages of the base relation of the virtual alpha). The formula for this case is:

$$TwoWayJoinCost(TRSize, n) = Pages(n) \times I/O_{weight} + TRSize \times Card(n) \times CPU_{weight}$$

The second case is when n is a stored alpha node with an index on a join attribute, or a virtual alpha node with an index on the join attribute on its base relation. The cost for this case is:

$$\begin{aligned} &TwoWayJoinCost(TRSize, n) = \\ &\quad TRSize \times Yao(Card(n), Pages(n), \lceil \frac{card(n)}{Card(n.joinAttr)} \rceil) \times I/O_{weight} + \\ &\quad TRSize \times \lceil \frac{card(n)}{Card(n.joinAttr)} \rceil \times CPU_{weight} + \\ &\quad TRSize \times \lceil \log_{fanout} Card(n) \rceil \times CPU_{weight} \end{aligned}$$

The cardinality of a join attribute $jAttr$ in the temporary result TR is estimated in the following way [5]: Let $jAttr$ be the attribute of a relation R and let $n = Card(R)$ and $b = Card(R.jAttr)$. Assuming uniform distribution of values, independence of value distribution in different columns, and random selection of values for the join attribute in TR from the original relation, estimation of $Card(TR.jAttr)$ can be reduced to the following statistical problem: Given n objects uniformly distributed over b colors, how many different colors are selected if one randomly selects t = Card(TR) objects? Bernstein et al [2] give an approximation of this value, shown here as the function $Estimate(n, b, t)$:

$$\begin{aligned} Estimate(n, b, t) &= t, \text{ for } t < b/2 \\ &= (t + b)/3, \text{ for } b/2 \leq t < 2b \\ &= b, \text{ for } t \geq 2b \end{aligned}$$

This function is used internally to estimate the cardinality of a join attribute in a temporary result. The same approach is also used to estimate the cardinalities of attributes in the α and β nodes also (except the ones that participate in a selection condition).

Finally, the cost of a P-node, i.e. the cost of the whole discrimination Network rooted at the P-node, is

given as follows:

$$Cost(P) = LocalCost(P) + \sum_{N \in children(\beta)} cost(N)$$

where

$$LocalCost(P) = \sum_{N \in children(P)} \{F_i(N) \times PerChildInsCost(N,P) + \{F_d(N) \times PerChildDelCost(N,P)\}$$

$PerChildInsCost(N,P)$ is essentially the same as $PerChildInsCost(N,\beta)$ except that there is no update cost here. Since a P-node does not store any tuples, $PerChildDelCost(N,P)$ involves no cost. This concludes the presentation of the cost functions. The discussion now turns to how these cost functions can be used to guide an optimizer in choosing a good discrimination network.

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. The number of Gator networks for a rule can be extremely large compared to the number of left-deep Rete networks. This can be seen by the fact that the number of left-deep binary trees is a subset of the set of binary trees, which is in turn a subset of the set of rooted trees with fanout of two or more at each level (like Gator networks).

To cope with the large search space for Gator networks, the Gator network optimizer implemented in Ariel uses a randomized state-space search technique. 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 [12], a problem that also has a very large search space. Early experiments were conducted [11, 19] which demonstrated that a randomized approach to Gator network optimization is superior to a dynamic programming (DP) approach like that used in traditional query optimizers [21]. With a DP approach, for rules with more than seven tuple variables, optimization time to build a Gator network exceeded several minutes. It was infeasible to optimize the network for rules with eight or more tuple variables. An approach to optimizing Gator networks needs to be able to produce good-quality results in a few minutes or less for up to 15 tuple variables. (This is the same as the limit on the number of tuple variables in an SQL SELECT statement in at least one major commercial DBMS product – more tuple variables than this are rarely needed in real applications).

Three randomized state-space search strategies were considered: iterative improvement (II), simulated annealing (SA) and two-phase optimization (TPO, 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 [14, 12, 13].

In the following discussion two sibling nodes in the discrimination network are said to be *connected* if

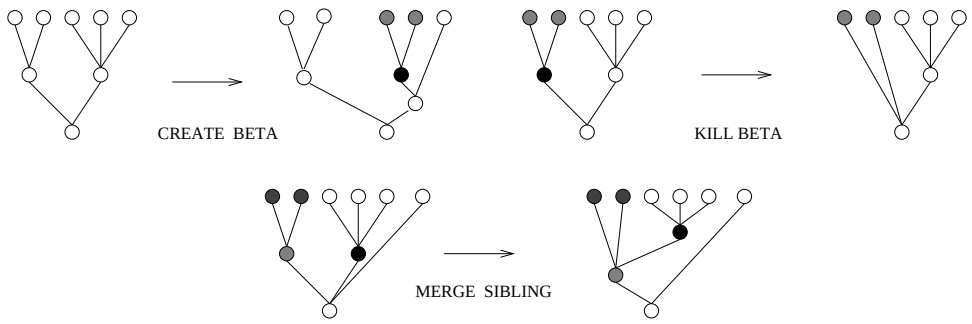


Figure 3: Local change operators.

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 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 3.

- *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 .

[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.

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

The optimizer implemented is capable of using II, SA and TPO. Each of the II, SA and TPO 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 TPO is given below.

4.1 Iterative Improvement

The Iterative Improvement (II) technique [23] 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 some 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. [16] for problems with many degrees of 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 decreases with time, the chances of an uphill move tend to zero [16, 14].

4.3 Two Phase Optimization

In its first phase, TPO 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

parameter	value
stopping condition	same time as that of TPO
local minimum	r-local minimum
next state	random neighbor

Figure 4: Parameters for II

parameter	value
initial state	random state
initial temp	2 * cost(initial state)
temp reduction	0.95*T_{old}
frozen	same as that of the SA phase of TPO

Figure 5: Parameters for SA

minimums. However, the low initial temperature makes climbing very high hills virtually impossible. It has been observed that TPO performs better than both II and SA approaches for optimizing large join queries [12].

4.4 Optimizer Tuning

The parameters used in this study for II, SA and TPO are given in Figures 4, 5 and 6 respectively. These parameters were chosen after extensive experimentation and by following guidelines given in the literature [1, 24, 23, 12]. TPO needed a lot of tuning effort compared to the other two algorithms. The performance of TPO depends on the performance of both the II and SA phases and hence more effort is needed to balance the two phases. Also, it was noticed that the performance of TPO is very sensitive to the initial temperature of the SA phase, in addition to the number of local optimizations of the II phase. For deciding the local minimum in II, the same approximation was used as by Ioannidis [12]. A state is considered to be an r-local minimum if the cost of that state is less than that of the cost of n randomly chosen neighbors (with repetition) of that state. In this paper, n was chosen to be the number of edges in the condition graph

parameter	value
stopping condition (II phase)	20 local optimizations
initial state	best of II ($best_{II}$)
initial temperature (T_0)	0.5*cost($best_{II}$), if cost($best_{II}$) $<$ 20000
	0.05*cost($best_II$), otherwise
equilibrium	Number of edges in the rule condition graph
temp reduction	0.95*T_{old}
frozen	temp $< T_0/1000$ and best state unchanged for 5 stages or
	total time >= (40*number of relations in rule condition) seconds

Figure 6: Parameters for TPO

of a rule. This is equivalent to the maximum number of β nodes in any Gator network for that rule and hence is an upper bound on the number of times a create-beta or a kill-beta can be applied. Deciding a local minimum by exhaustively searching the neighbors of a state is an expensive process and hence we believe that the choice of using an r-local minima is a more practical one.

The optimizer in this study creates a complete new Gator network every time it applies a local change operator. This is an inefficient way to move from one state to another. An alternative is to directly modify the data structure representing the state to move it to the next state, and to undo these changes if they are not beneficial. Moreover, the Exodus eg++ compiler [20], an unsupported compiler that is not of commercial quality, was used. The Gator network data structures were implemented using "dbclasses" [20] rather than regular C++ classes, further slowing optimizer performance, since dereferencing a pointer to a dbclass object takes many instructions instead of one. The E compiler was used to reduce total coding effort, since the Gator networks must be made persistent, and dbclasses provided persistence without writing extra code. Due to these performance limitations of our current implementation, we are confident that the Gator optimizer could easily be speeded up by a factor of 10 or more without changing the algorithms used for search.

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.

In the original Ariel system, there were seven different types of α nodes with slightly different behavior [7]. 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.

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.

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. An important objective is to choose a join plan with the minimum cost. However, since choosing token join plans must be done very frequently while finding an optimized Gator network for one trigger, it is too expensive to use traditional query optimization [21] 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 the smallest connected sibling. This gives a reasonable join order plan quickly.

6 Optimizer Characteristics and Performance

This section presents the details of various experiments conducted to study the relative behavior of II, SA and TPO for various rules under different update frequency distributions, catalogs and cost models. Rules were created on synthetically generated databases, which had the following properties:

	Relation Cardinality	% of unique values in attributes
Catalog 1	[1000, 100000]	[90, 100]
Catalog 2	[10, 100] - 20% [100, 1000] - 64% [1000, 10000] - 16%	(0, 20) - 70 % [20, 100) - 5% 100 - 25%
Catalog 3	[1000, 10000]	[90, 100]

The table gives the cardinality distribution for tables in each catalog. Every table has a primary key attribute. For the other attributes, the table shows the percentage of attributes which fall in each cardinality range. E.g. for Catalog2, 64% of the tables have between 100 and 1000 tuples. Furthermore, 5% of the attributes have at least 20% and fewer than 100% as many unique values as the primary key.

Indices were created only on large relations in the first database. 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.

Star type One relation participates in a join with all the other relations in the rule condition.

Random type Joins between relations are chosen randomly to create a connected rule condition graph with no cycles.

Here are examples of rules with the three types of RCGs:

String Type	**Star Type**	**Random Type**
define rule Rule1	define rule Rule2	define rule Rule3
if R1.a = R2.b	if R1.a = R2.b	if R1.a = R2.b
and R2.c = R3.d	and R1.c = R3.c	and R1.c = R3.c
and R3.d = R4.e	and R1.b = R4.d	and R1.b = R4.d
then <action1>	then <action2>	and R4.d = R5.e
		then <action3>

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.

Even All the relations have the same update frequency.

Step The update frequencies of relations decrease in a stair-like manner.

The actual frequencies used are summarized in the table below. In all cases, frequencies sum to one.

	Equal	Step	Skew
5 relations	0.2 each	0.4, 0.3, 0.2, 0.05, 0.05	0.8, 0.05 others
10 relations	0.1 each	0.4, 0.3, 4 each with 0.05 and 0.025	0.7, 0.124, 0.022 others
15 relations	0.067 each	0.3, 0.2, 0.14, 0.04, 0.03, and 4 each with 0.02	0.6, 0.14, 0.02 others

The *size* of a rule is the number of tuple variables in its condition. Rules of size 5, 10 and 15 were created with string, star and random type rule condition graphs. Each one of these rules was tested with equal, step and skewed frequencies, with cost models CM1 and CM2 and catalogs Catalog1 and Catalog2. For a number of (RCG, size, frequency, cost model, catalog) combinations, each of TPO, SA and II was run 10 times with different random seeds. For each algorithm, the average of the output state in all the 10 runs was computed. II was run the same amount of time as was taken by TPO. For each of these cases and for each algorithm, the average scaled cost was computed by dividing the average cost of 10 runs of that algorithm by the best state found by all the runs of all the algorithms for that case.

Due to lack of space, only a few interesting graphs that exhibit the behavior of II, SA and TPO are shown in Figure 7. Results are shown for Catalog1 and Catalog2 and both cost models for a rule with star type RCG. Also, results are shown for a rule with string type RCG with equal frequency for Catalog1 and CM1, and for a rule with random type RCG with skewed frequency for Catalog2 and CM2.

It can be seen that for rules with size 5, irrespective of the catalogs, cost models and frequencies, all the algorithms are doing well. There is no difference in the costs of states produced by different algorithms. As

the rule size increases from 5 to 15, the difference in the relative behavior of the algorithms also increases. Also, as the rule size increases, the average behavior of the algorithms tends to go away from 1, i.e. the algorithms become less stable. In general, TPO performs better than II and SA and there is no clear winner between II and SA. Both II and SA exhibit worst case behavior in some cases. No significant difference was observed in the behavior of the algorithms for the cost models CM1 and CM2. It was observed that when considering the best of all runs in the experiments, all of II, SA and TPO performed well.

Graphs D, E and F in Figure 7 show interesting cases concerning the relative behavior of the three algorithms. In graph F, all the three algorithms are doing well and their average behavior is very close to 1 (similar to the behavior of the algorithms for rules with size 5). Graphs D and E show cases where II and SA are doing better than TPO. These graphs also illustrate the difficulty in tuning TPO to perform well in all cases. The behavior of the algorithms in these graphs is explained next and some general conclusions about the overall behavior of the algorithms are then given.

In graph D, II is performing slightly better than TPO. Here, the problem is in deciding the crossover point between II and SA in TPO. This decision is crucial, especially when the search space contains many local minima at high-cost states with a small but significant portion of them at low-cost states (space A2, similar to the search space of left deep query trees in [13]). In this case, doing a few iterations in the II phase of TPO might leave the starting state of SA at a high-cost state (because there are many local minima at high-cost states) making the overall result of TPO not satisfactory. Doing more iterations or local optimizations in II is always beneficial in this case, because that helps to find a low-cost local minimum. The presence of many high-cost local minima also explains the behavior of SA in this case. SA searches high-cost states when the temperature is high, and when the temperature gets low, it reaches a local minimum state and searches around that state. Since there are many local minima at high-cost states, SA also can get trapped in one of the high-cost states and hence its performance is not good. In this case, if the number of iterations in the II phase of TPO is increased, then TPO is going to do at least as well as II.

In graph E, SA is doing better than TPO and II is doing worse than TPO. Here, partly, the problem is in estimating the initial temperature of the SA phase of TPO. Here, many runs of II generate a high-cost local minimum and hence its performance is not good. TPO seems to extricate itself out of these high cost states in many of the cases but not all. The reason seems to be that the cost of states separating the low-cost states is not very low and hence the initial temperature ($0.5*best_{II}$, here $best_{II} < 20000$) seems to be not enough to jump over those states. Also, the low value of SA shows the presence of low cost local minima. In fact, for this case, when we repeated the experiments with high initial temperature ($1.0*best_{II}$) the average value of TPO came down to 1.307 compared to the current value 1.405 and the SA value of 1.230. In general, we noticed that the behavior of TPO is very sensitive to the initial temperature and we tuned it carefully for various cases. Here, part of the reason that TPO is not doing well could also be due to the existence of many local minima at high cost states. This is because the average behavior of SA is also not close to 1 which suggests the possibility of many high-cost local minima.

In general, it can be stated that TPO does well and the performance of II and SA are close to TPO (within 10-20% in most cases). A possible explanation for this is that the search space, in general, contains most of the local minima at low-cost states with reasonably high cost states separating the local minima.

Each iteration of II generates a random state and follows downhill moves until it reaches a local minimum. Since most of the local minima are at low-cost states, II is able to find a good local minimum. SA explores high-cost states when the temperature is high and reaches the local minima states at low temperatures and searches around those states. Since, again, there are many local minima at low states, it is able to find a good one in most of the cases. TPO starts with a good local minimum state and performs search starting with low initial temperature. It seems that, since the temperature is low, it is able to search or come across a lot of low-cost states and hence it has a high chance of finding a better state. Also, in other experiments, it was noticed that the SA phase of TPO does not do well when the starting temperature is very low. Based on this, coupled with graph E, it can be said that the cost of states separating the low-cost states in reasonably high (for states with cost < 20000, the SA phase of SA never performs well when the starting temperature is less than $(0.5 * best_{II})$).

In graph B, the performance of SA is worse compared to II and TPO. II is doing well in this situation, which means that there are enough valleys containing a low cost minimum so that II can almost always find a good overall solution. SA seems to be getting trapped in high-cost states and does not reach the low-cost local minimum states. The solution space in this case seems to contain high-cost valleys and the temperature does not seem to be high enough to escape these valleys.

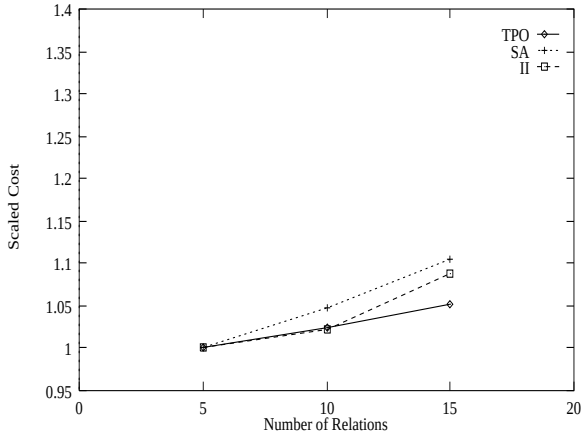
Even though II and SA are performing close to TPO in most of the cases, the ability of TPO to avoid worst-case behavior makes it the winner. It is used in the next section for generating the optimal Gator network to compare with optimal Rete and TREAT.

Tables showing the average optimization time in seconds taken by TPO and SA for all the cases are shown in Figure 8. The time taken by II is not shown here because it was given the same amount of time as TPO. Except in a few cases, TPO takes less time than SA. The difference in the time taken by TPO and SA increases as rule size rises from 5 to 10. At size 15, both II and SA take almost the same time. Again, no significant differences were found between the optimization times of the two cost models.

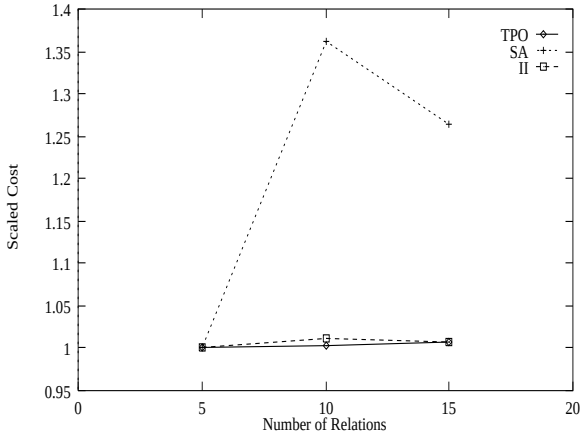
7 Performance Evaluation and Optimizer Validation

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. This 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).

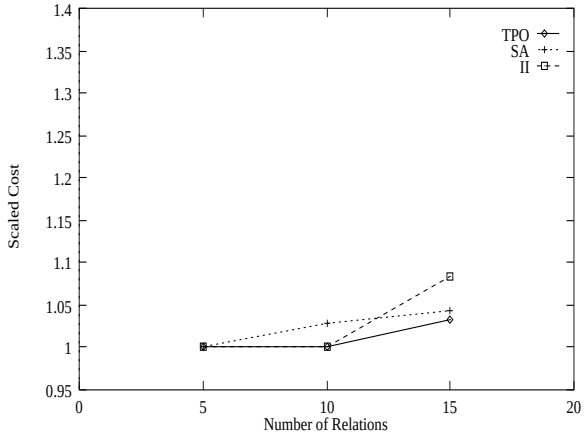
The Ariel active relational DBMS was used as a testbed for conducting all the experiments. The average rule activation time was measured by processing a randomly generated stream of updates. The table to which



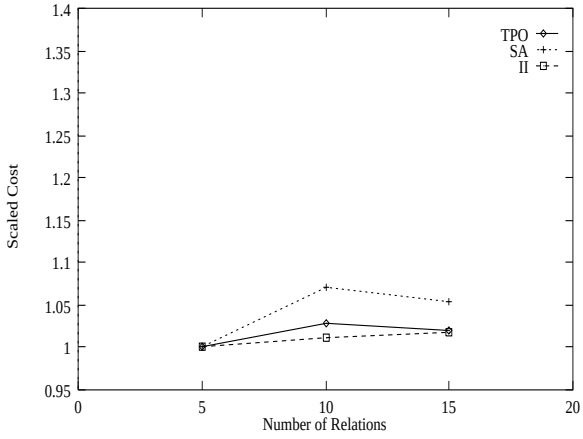
(A) Star type RCG with Step Frequency Distribution, CM1, Catalog 1



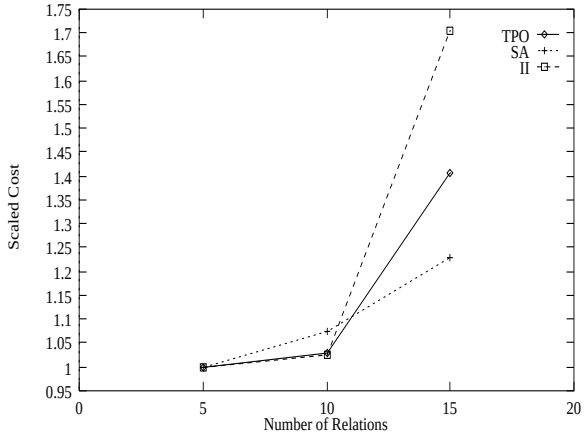
(B) Star type RCG with Step Frequency Distribution, CM1, Catalog 2



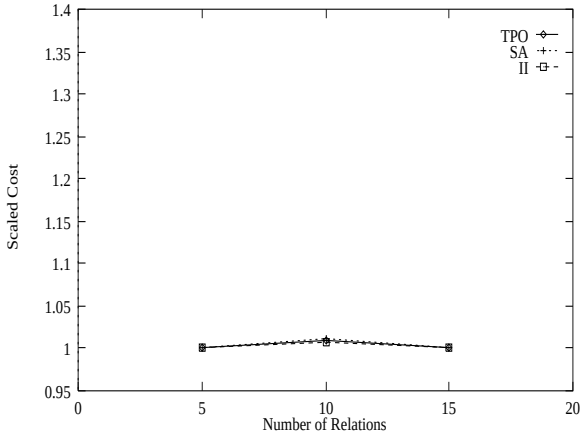
(C) Star Type RCG with Step Frequency Distribution, CM2, Catalog 1



(D) Star Type RCG with Step Frequency Distribution, CM2, Catalog 2



(E) String Type RCG with Eq Frequency Distribution, CM1, Catalog 1



(F) Random Type RCG with Skew Frequency Distribution, CM2, Catalog 2

Figure 7: Average scaled cost of the optimal Gator network found by SA, II and TPO

	star,step,CM1, Catalog1			star,step,CM1, Catalog2			star,step,CM2, Catalog1		
Size	5	10	15	5	10	15	5	10	15
TPO	39	215	598	29	196	600	36	230	600
SA	47	247	600	28	201	537	44	243	600
	star,step,CM2, Catalog2			string,eq,CM1, Catalog1			random,skew,CM2, Catalog2		
Size	5	10	15	5	10	15	5	10	15
TPO	40	237	600	33	191	530	39	209	583
SA	37	271	600	40	219	556	48	204	587

Figure 8: Average optimization time (in seconds) of TPO and SA.

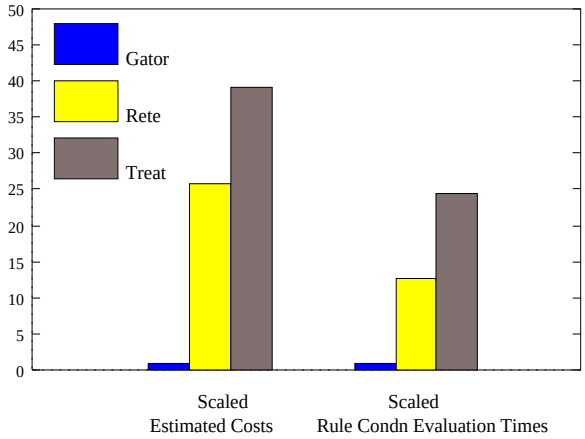
an update was applied was determined using a frequency distribution equivalent to the update frequency statistics maintained in the system catalog. Inserts were done on each table, and the token testing time for each was measured. Then, a "total rule condition testing time" was calculated by multiplying the time spent propagating a token for each table by the update frequency for the table.

For all the tests discussed, an optimized Gator network is compared with a TREAT network (for which there is only one choice) and an *optimized,* left-deep Rete network. The Rete networks were optimized using a dynamic programming-style Rete network optimizer. Optimized Rete networks were considered in order to give a fair comparison between Gator and Rete.

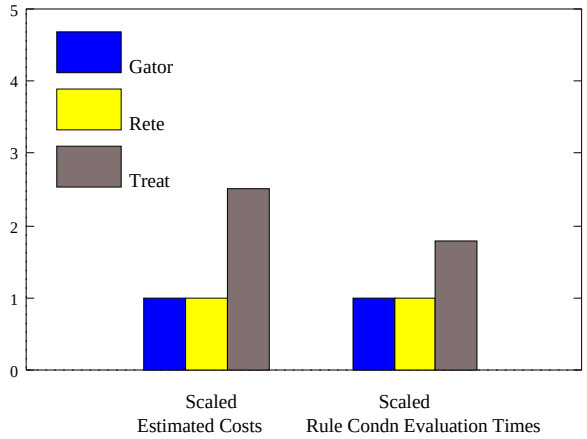
The different parameters that were varied were database statistics, number of tuple variables in the RCG, the shape of the RCG, the placement and number of selection conditions in the rules, and the frequency distribution of updates. Since the relation sizes are small in Catalogs 2 and 3, no indexes were created. For Catalog 1, B^+-tree indexes were created on primary keys of large relations. As it is not possible to show all the results here, a sample of the results of various experiments is given below. Due to the fact that Exodus does not allow buffer space to be set to be very low, and it does an excellent job of buffer management, it was not possible to truly test token propagation cost for cost model 2. All tests were run on a Sun SPARCstation 5/110.

Figures 9 and 12 show relative estimated vs. actually measured token propagation times for Gator, Rete and TREAT for various parameter combinations. Figure 9 (A) and (B) show the results for a rule with five tuple variables, with string and star type RCGs and step frequency distribution. It can be seen that Gator is doing much better than Rete and TREAT in (A). In (B), the Gator network which the optimizer comes up with is the same as the optimal Rete network. This shows the flexibility of Gator – when Rete or TREAT is best, Gator will normally arrive at that result.

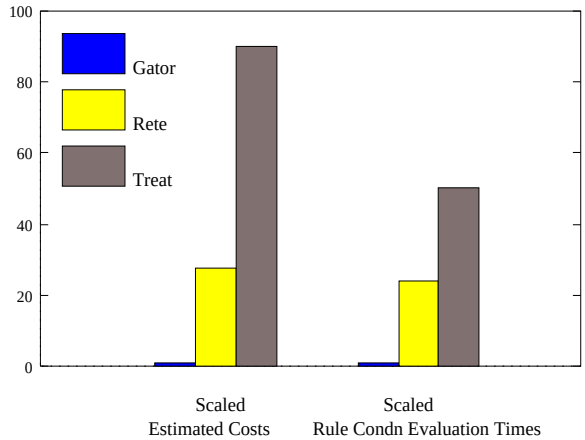
Figure 9 (C) and (D) show the results for a rule with ten tuple variables, with random type RCG and step frequency, with two different databases. The RCG for the rule used was as shown in Figure 10 (A). It can be seen that in both the cases, the Gator network outperforms both Rete and TREAT networks. However,



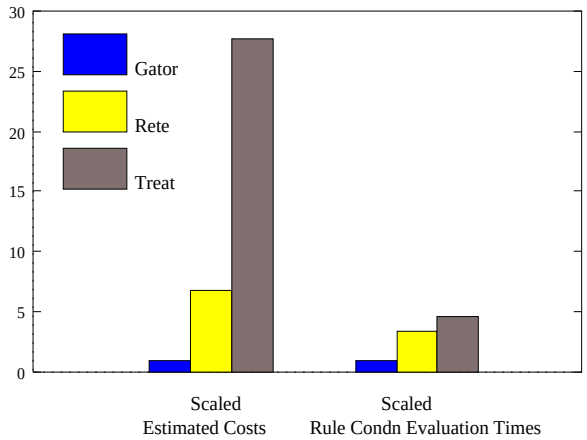
(A) String type RCG with step frequency distribution.



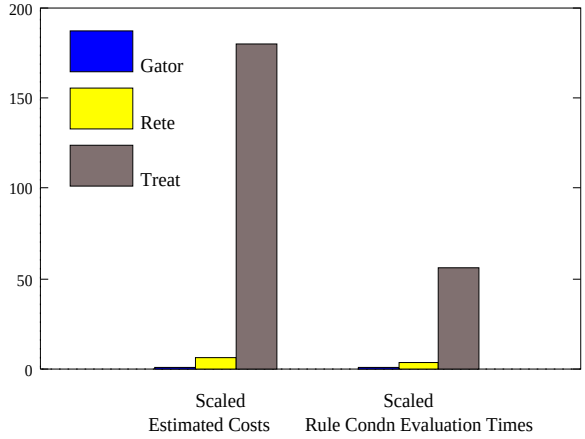
(B) Star type RCG with step frequency distribution.



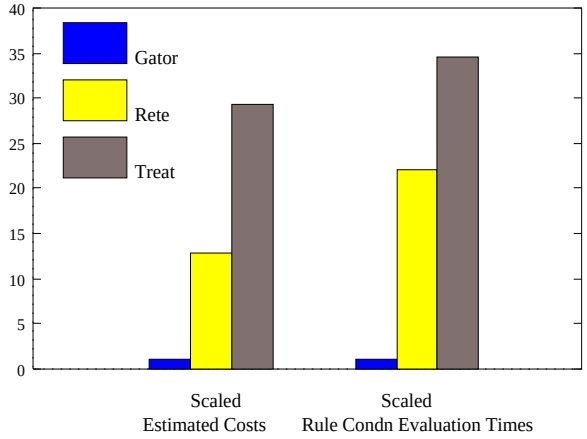
(C) Random type RCG with step frequency distribution, using Catalog 2.



(D) Random type RCG with step frequency distribution, using Catalog 3.

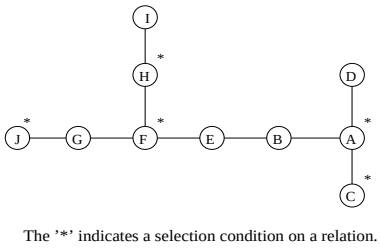


(E) String type RCG with equal frequency distribution, using Catalog 2.

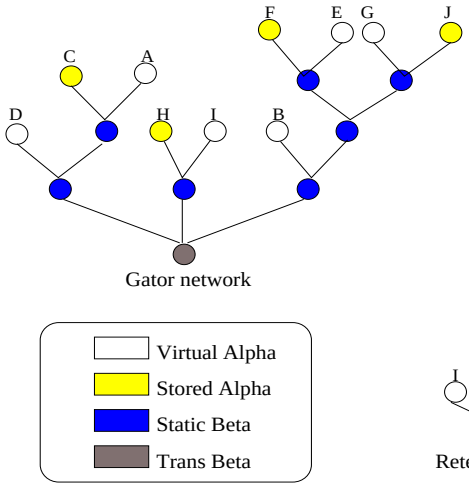


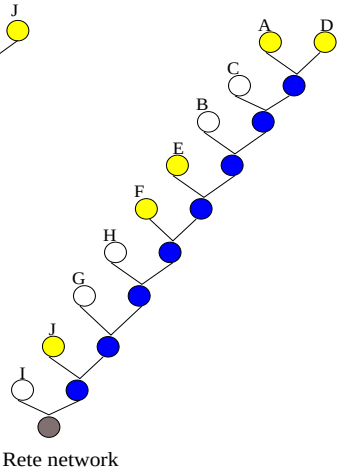
(F) String type RCG with equal frequency distribution, using Catalog 3.

Figure 9: Comparison of estimated costs and rule condition evaluation times.



(A) Random type RCG with ten relations.





(B) Gator and Rete networks.

Figure 10: A random type RCG of size ten and its optimal Gator and Rete networks.

it may be noticed that the estimated costs and the actual rule condition evaluation times are proportional only in the case of (C), whereas in (D) the Gator optimizer overestimates the cost of the TREAT network. During the tests, it was observed that this was so in a few cases. We believe that this is most likely due to over-estimation of the size of temporary results when a token is joined across a set of nodes in TREAT. Getting accurate estimates for the size of temporary results is challenging because of the way errors in join selectivity compound during a long join sequence.

The Gator and Rete networks generated for (D) are shown in Figure 10 (B). 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 the case of the Gator network, the relations with high update frequencies (D=0.4 and I=0.3) were pushed down the discrimination network, towards the P-node. This means that fewer token joins need to be done as tokens propagate through the network due to updates. Also, the stored α nodes with low size were at the top of the network which helps to reduce the size of the β nodes below them. Another interesting observation was that the optimizer was clever enough to never form a β node with two virtual alpha nodes as children, since that could potentially increase the size of the β node. Along these lines, it can be seen from Figure 10 (A) that the relations B and E do have a join edge between them. Neither of them has a selection condition on it. They could have been made the children of the same β node, however the optimizer pruned out that case.

In the case of the Rete network, relation I was closest to the P-node, which one would expect to be beneficial. However, D was at the top. Intuitively, one would expect this to create a higher-cost network. However, we forced the creation of a network with both I and D near the bottom, and both the predicted and the actual costs were higher than that of the network chosen by the optimizer. This illustrates the value of using optimization for discrimination networks – there are so many competing cost factors that "obvious"

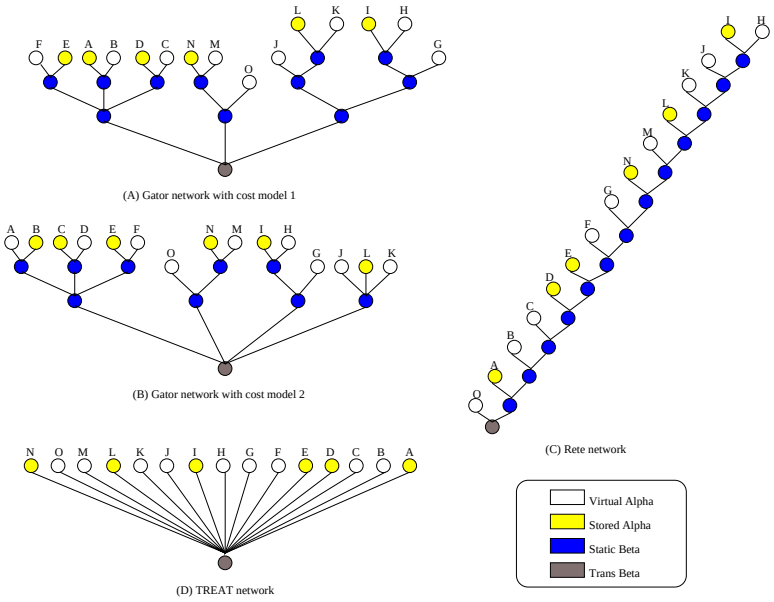


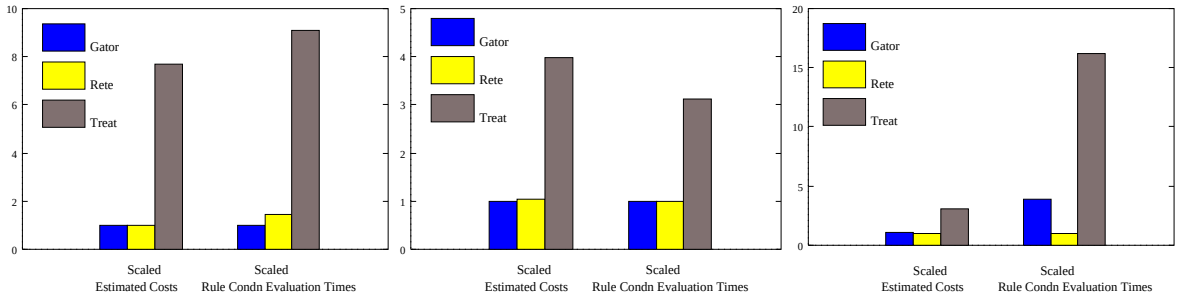
Figure 11: Gator, Rete and TREAT networks.

improvements may not really be beneficial. In the case of TREAT, whenever a new token enters the network it always has to participate in a join across the other α nodes, which explains its higher rule condition testing time.

Figure 9 (E) and (F) show the results for a rule with fifteen tuple variables, with string type RCG and equal frequency distribution, with two different databases. Again, it can be observed that in both the cases, Gator and Rete perform much better than TREAT, and in one case, the Gator network does much better than the Rete network as well. The Gator, Rete and TREAT Networks generated are shown in Figure 11. It can be seen in Figure 11 that the optimal Gator networks found by the Gator optimizer for the two different cost models differ, in that the optimizer decides to have fewer β nodes for cost model 2 (where I/O cost is significant). The reason is that the cost of maintaining β nodes is higher than the cost for performing a join between more nodes in this case. It was also observed for this case that the cost of the optimal Rete network was higher than that of the TREAT network.

A Rete network is constrained to be a left-deep binary tree. Since the frequency distribution is equal, the Rete network loses out to the Gator network on the whole, since it takes a lot more time for rule condition evaluation in the case of updates that are made to relations which are at the top of the network. However in the Gator network, the time taken for rule condition evaluation is more or less the same for most of the relations.

Figure 12 (A) and (B) show the results for a star type rule with fifteen tuple variables and step frequency distribution with two different databases. It can be seen that the performance of Gator and Rete networks was comparable *as predicted* by the estimated costs. And again, both these networks outperformed the



(A) Star type RCG with step frequency distribution, using Catalog 2.

(B) Star type RCG with step frequency distribution, using Catalog 3.

(C) Star type RCG with step frequency distribution, using Catalog 1.

Figure 12: Comparison of estimated costs and rule condition evaluation times.

TREAT network.

Figure 12 (C) shows the results for a star type rule with ten relations and step frequency distribution. In this case, indexes were created on large relations. It can be seen from the graphs that the Rete network did slightly better than the Gator network as the cost formulae had correctly predicted. This means that the randomized Gator optimizer was not able to arrive at the exact optimal Rete as the optimal Gator. Occasional errors like this are to be expected in a randomized optimization algorithm. However, it was observed during all the various tests that were run that this was a very rare occurrence. In this case, one point in favor of the Gator network optimizer was that it took about half the time which the Rete optimizer took to come up with the optimal Gator network. In general, it was noted that the time needed for obtaining an optimal Rete network was large in the case of rules with star type RCG.

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 conclusion 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.

8 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, but not a full complement of them, i.e. it is neither a TREAT network nor a Rete network. When I/O cost is significant, Gator is clearly better than both Rete and TREAT. When I/O cost is low, e.g. in an environment with plentiful buffer space, the Gator networks

generated have very few if any β nodes with more than two inputs. Hence, for environments like this, optimized bushy Rete networks would do approximately as well as optimized Gator networks. Optimized left-deep Rete networks are only competitive with Gator networks when update frequency is at least partly skewed and I/O cost is low. Optimized Gator networks virtually always have short root-to-leaf paths. They are balanced, and the fanout of a β node is typically in the range two to four. The short root-to-leaf path length reduces the number of nodes that need to be touched when tokens are processed. Moreover, as much as possible, the Gator optimizer avoids materializing large β nodes, which saves time both for rule condition testing and for β node update. Overall, it is clearly 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 many joins in active databases. This study has gone beyond merely an optimizer simulation to actually validate the results of the optimizer during rule condition testing. Optimizer validation is rarely attempted due to the difficulty of doing the tests, as well as showing strong correlation between expected and actual results. An exception is the paper by Mackert and Lohman [17]. The work given here can help make it possible to implement the capability to efficiently and incrementally process triggers with large number of joins in their conditions in commercial database systems, thus making a new, powerful tool available to database application developers. Finally, the same implementation of optimized Gator networks could be used to optimize maintenance of materialized view. This would provide fast materialized view maintenance essentially "for free" in terms of DBMS implementation complexity.

References

[1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines*. John Wiley and Sons, 1990.

[2] P. A. Bernstein, N. Goodman, E. Wong, C. L.Reeve, and J. James B.Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 6(4):602–625, December 1981.

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

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

[5] S. Ceri and G. Pelagatti. *Distributed databases*. McGraw-Hill computer science series, 1984.

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

[7] E. 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.

[8] E. N. Hanson, M. Chaabouni, C. Kim, and Y. 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.

[9] E. N. Hanson, I.-C. Chen, R. Dastur, K. Engel, V. Ramaswamy, C. Xu, and W. Tan. Flexible and recoverable interaction between applications and active databases. *VLDB Journal*, 7(1), 1998. In press.

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

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

[12] Y. Ioannidis and Y. C. 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.

[13] Y. Ioannidis and Y. C. 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.

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

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

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

[17] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. Technical Report RJ4989, IBM Almaden Research Center, January 1986.

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

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

[20] J. E. Richardson, M. J. Carey, and D. T. Schuh. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15(3), 1993.

[21] 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 [22]).

[22] M. Stonebraker, editor. *Readings in Database Systems*. Morgan Kaufmann, 1994.

[23] A. Swami and A. Gupta. Optimization of large join queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 8–17, 1988.

[24] P. J. M. van Laarhoven and E. H. Aarts. *Simulated Annealing: Theory and Applications*. D. Reidel Publishing Company, 1987.

[25] Y. Wang and E. 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.