# Optimized Rule Condition Testing in Ariel using Gator Networks*

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

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

23 October 1995

TR-95-027

**Abstract**

This paper presents a new 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, and is designed as a target for a discrimination network optimizer. Ariel now has an optimizer that can choose an efficient Gator discrimination network for testing the conditions of a set of rules, given information about the rules, database size and attribute cardinality, and update frequency distribution. The optimizer uses a randomized strategy similar to one which has been successfully used previously to optimize large join queries. Use of Gator gives large speedups (3 times for one realistic rule tested – potentially much more) compared with the unoptimized TREAT strategy formerly used in Ariel.

## 1 Introduction

A crucial component of an active database system is the mechanism it uses to test rule conditions as the database changes. Tools for artificial intelligence programming known as production systems, such as OPS5 [2], use structures called discrimination networks for rule condition testing. For a number of years, the Rete [3] and TREAT [15] discrimination network algorithms have been used in production system implementations. Variations of TREAT have also been used in the Ariel [4, 6] and DATEX [1] active database projects. The difficulty with the standard Rete and TREAT algorithms is that they do not provide a way to optimize Rule condition testing based on database size, predicate selectivity, and update patterns. In response to this observed need for optimization, we have developed a new, generalized discrimination network structure called Gator (Generalized Treat/Rete).

Gator networks are general tree structures. Rete and TREAT networks are special cases of Gator. The leaf nodes of Rete, TREAT and Gator networks are $\alpha$-memories that hold data matching single-relation selection predicates. In Gator, $\beta$ nodes that hold intermediate join results can

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

have two or more inputs, not just two as in Rete. TREAT networks have no $\beta$ nodes. Gator networks are suitable for optimization because there are a very wide variety of Gator structures that can perform pattern matching for a single rule. In contrast, there is only a single TREAT network for a given rule, and Rete networks are limited to binary-tree structures.

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 $\beta$ nodes usually is greater than their benefit in Rete. 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 OPS5 programs [12]. But even optimized Rete networks still have a fixed number of $\beta$ nodes, which take time to maintain and use up space. With Gator, it is possible to get additional advantages from optimization, since $\beta$ nodes are only materialized when they are beneficial.

This paper describes the Gator algorithm, outlines a cost model for Gator networks, and presents how the Gator optimizer and rule condition matching algorithm have been implemented in a modified version of Ariel [14]. Performance figures are given that demonstrate a substantial speedup in the rule condition testing performance of Ariel.

## 2 The Gator Algorithm

To begin illustrating Gator 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 rule 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
```

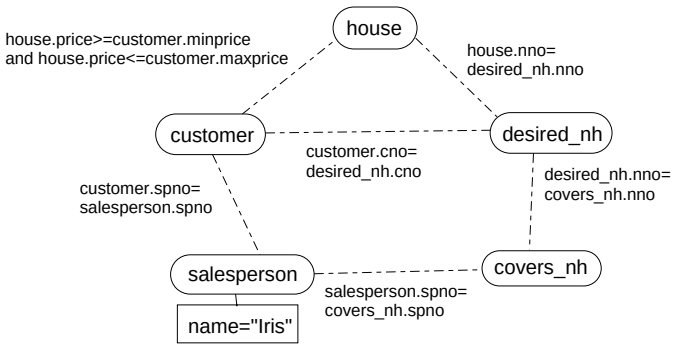



Figure 1: A rule condition graph for IrisRule.

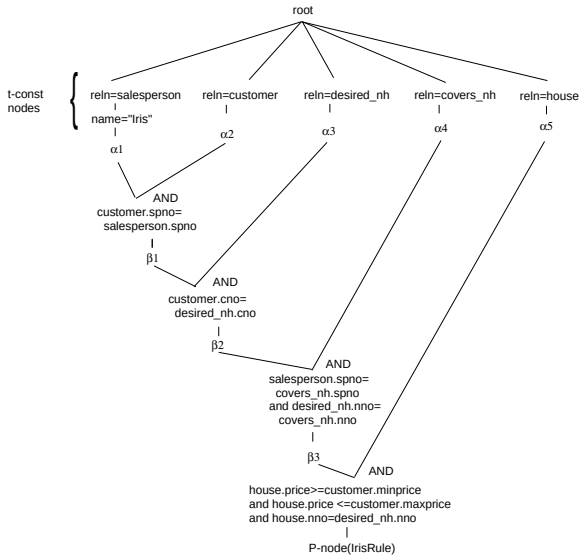



Figure 2: A Rete network for the rule IrisRule.

```
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. A Rete network for this rule is shown in figure 2. A sample TREAT network for the rule IrisRule is shown in figure 3. An example Gator network for the rule IrisRule is shown in figure 4. Gator networks use "+" tokens to represent inserted tuples or objects, and "−" tokens to represent deleted objects, just as in Rete and TREAT. Modified tuples are treated as deletes followed by inserts. As an example of Gator matching, suppose that a new customer for Iris is inserted. This would cause the creation of a "+" token $t_1$ containing the new 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 packaged as a + token and joined with $\alpha 3$. If elements of $\alpha 3$ join with this token, each joining pair is in turn 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. After joining to $\alpha 4$, the results would

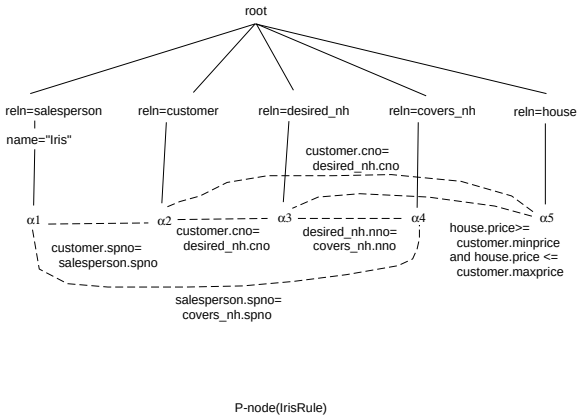



Figure 3: A TREAT network for the rule IrisRule.

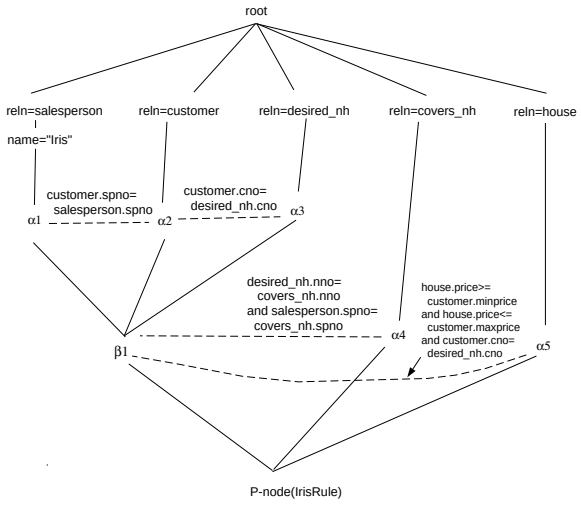



Figure 4: A Gator network for the rule IrisRule.

be joined next to $\alpha 5$, and vice versa. In Gator, as in Rete, the root $\beta$ node for a rule is called the P-node, and if a token arrives there, that signals that the rule is to be triggered. 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 for the new combination.

This example gives the flavor of how active database rule condition matching is performed by propagating information through a Gator network. A detailed description of how Gator does pattern matching by propagating tokens is not given here due to lack of space, but is presented elsewhere [5].

# 3 Cost Functions

So the optimizer can choose between different Gator networks, a cost estimation function is needed to evaluate the expected cost of a network. Our cost function estimates the run time cost associated with propagating tokens through the network, including the cost of:

- Performing selections and joins on tokens.
- Updating the contents of the storage structures in the discrimination network with the results of these selections and joins.

The cost is a weighted measure of I/O and CPU utilization. The cost function for a Gator network node is defined recursively. This means that the cost of a node represents the cost of the subnetwork rooted at that node and involves the cost of all its descendants all the way up to the leaf nodes. The parameters used in developing the cost model are given in the table showin in figure 3.

## 3.1 Cost functions for the $\alpha$ nodes

The size of an $\alpha$ node is estimated as the product of the size of the relation from which the $\alpha$ node is derived, and the selectivity of the associated selection predicate.

$$Card(\alpha) = S(R(\alpha)) \times Sel(\alpha)$$

The insert frequency $F_i(\alpha)$ and delete frequency $F_d(\alpha)$ of an $\alpha$ node are defined by the following equations:

$$F_i(\alpha) = F_i(R(\alpha)) \times Sel(\alpha)$$

$$F_d(\alpha) = F_d(R(\alpha)) \times Sel(\alpha)$$

At present there are no indexing facilities for $\alpha$ and $\beta$ nodes in the Ariel DBMS. In order to insert a tuple we need one disk read and one disk write. The cost of inserting a tuple into an $\alpha$ node can therefore be given by,

$$C_i(\alpha) = (CPU_{weight} + 2 \times IO_{weight}) \times F_i(\alpha)$$

The absence of indices means that in order to delete a tuple all the tuples of the $\alpha$ node need to be scanned. After deleting the tuple, the page that contained the deleted tuple needs to be written back to the disk. This results in a deletion cost given by,

$$C_d(\alpha) = [\ CPU_{weight} \times Card(\alpha) + \ IO_{weight} \ \times \ \{\frac{Card(\alpha) \times bytePerTuple(\alpha)}{bytePerPage} + 1\}] \ \times F_d(\alpha)$$

The total cost of an $\alpha$ node, $C_t(\alpha)$ is given by,

$$C_t(\alpha) = C_i(\alpha) + C_d(\alpha)$$

| | |
|---|---|
| *bytePerPage* | The page size in bytes. This is a system specific value. We have assumed that the pages are of size 4K bytes. |
| $CPU_{weight}$ | The relative weight for CPU utilization. This indicates the time to insert/delete a tuple in a node or evaluating a selection predicate of a tuple or a join predicate between a pair of tuples. This value is normalized to 1. |
| $I/O_{weight}$ | The relative weight of an I/O operation. By default, the $CPU_{weight}$ to $I/O_{weight}$ ratio is 1:15. |
| $N$ | A node in the discrimination network. Could be any of the following: $\alpha$, $\beta$, or a P-node (also known as a trans-$\beta$). |
| *bytePerTuple(N)* | Tuple size of node N in bytes. |
| *tuplePerPage(N)* | The number of tuples per page. This is equal to $\lfloor \frac{bytePerPage}{bytePerTuple(N)} \rfloor$ |
| $F_i(N)$ | The insert frequency of $N$, i.e., the frequency of the arrival of "+" tokens at N. |
| $F_d(N)$ | The delete frequency of $N$, i.e., the frequency of the arrival of "−" tokens at N. |
| $Card(N)$ | The Cardinality of $N$. Indicates the number of tuples in $N$. |
| $Pages(N)$ | The Number of Pages occupied by $N$. |
| $C_i(N)$ | The insertion cost. Indicates the cost of inserting a tuple into node $N$. |
| $C_d(N)$ | The deletion cost. Indicates the cost of deleting a tuple from node $N$. |
| JSF($N_i$,$N_j$) | The JSF (Join Selectivity Factor) of a join between two nodes $N_i$ and $N_j$. This is the fraction of the tuples in the cartesian product of $N_i$ and $N_j$ that satisfy the join condition between $N_i$ and $N_j$. |
| $Cost(N)$ | The cost of the subnetwork rooted at N. This is the sum total of the costs of the children of $N$, the cost of performing joins for the tokens arriving at all the children of $N$ and the cost of updating the contents of the node $N$. |
| $LocalCost(N)$ | The cost of the local processing required for $N$. This includes the cost of performing joins for the tokens arriving at all the children of $N$ and the cost of updating $N$. |
| $R$ | A relation defined in the database catalog. |
| $S(R)$ | The cardinality of relation $R$. |
| $F_i(R)$ | The insertion frequency of relation $R$. This is the ratio of the number of tuples inserted into the relation $R$ to the number of tuples inserted into all the relations in the database. |
| $F_d(R)$ | The deletion frequency for the relation $R$. This is the ratio of the number of tuples deleted from the relation $R$ to the number of tuples deleted from all the relations in the database. |
| R($\alpha$) | The relation from which the $\alpha$ node is derived. |
| Sel($\alpha_i$) | The selectivity factor of the selection predicate associated with the $\alpha$ node $\alpha_i$. Indicates the fraction of the tuples in R($\alpha$) that satisfy the selection predicate. |
| leaves($\beta$) | The set of $\alpha$ nodes that form the leaves of the subnetwork rooted in the $\beta$ node. |
| $\prod_S$ | Symbol to represent the product of the sizes. |
| $\prod_\sigma$ | Symbol to represent the product of selectivity factors. |
| $\prod_\bowtie$ | Symbol to represent the product of join selectivity factors. |
| $\Psi$ | Symbol to represent product of $\prod_\sigma$ and $\prod_\bowtie$ |


Figure 5: Different $\beta$ nodes with the same leaf set, like those shown, may have different costs, but have identical size and update frequency.

### 3.2 Cost Functions for $\beta$ Nodes

A $\beta$ node in the Gator network can have two or more children. These child nodes in turn might be $\beta$ nodes and so on. When a token arrives at a child of a $\beta$ node, it participates in a multi-way join with all the siblings of this child and the resulting token set is fed into the parent $\beta$ node. This implies that the size, cost and update frequencies of a $\beta$ node are governed by its children. The cost of a $\beta$ node is dependent on the shape of the subnetwork rooted at that $\beta$ node. On the other hand, *the size and update frequencies of a $\beta$ node are independent of the shape of the subnetwork rooted at that $\beta$ node*. In other words, for a given set of leaf nodes of a subnetwork, irrespective of the way in which these leaf nodes are joined, the size and insert and delete frequencies of the root $\beta$ node of the subnetwork should be the same, as illustrated in figure 5.

The objective is to come up with an estimate for the size and update frequency of the $\beta$ node that is independent of the shape of the subnetwork rooted in that $\beta$ node. For the calculation of the size and the update frequencies of a $\beta$ node, we define the following parameters.

1. $\prod_S(\beta)$: This parameter is the size of the cartesian product of the relations associated with the $\alpha$ nodes in leaves($\beta$).

$$\prod_S(\beta) = \prod (S(R_k)) , \quad where\ R_k = Reln(\alpha_k)\ and\ \alpha_k \in leaves\ (\beta)$$

2. $\prod_\sigma\ (\beta)$: This parameter is the product of the selectivity factors of the selection predicates of the $\alpha$ nodes in leaves($\beta$).

$$\prod_\sigma(\beta) = \prod (Sel(\alpha_k)) , \quad where\ \alpha_k \in leaves\ (\beta)$$

3. $\prod_\bowtie\ (\beta)$: This is the product of the JSFs associated with all the edges between $\alpha$ nodes in leaves($\beta$) (the edges correspond to join edges in the rule condition graph).

$$\prod_\bowtie(\beta) = \prod (JSF\ (\alpha_i, \alpha_j)) , \quad where\ \alpha_i,\ \alpha_j \in leaves\ (\beta)\ and\ \exists edge(Reln(\alpha_i), Reln(\alpha_j))\ in\ the\ rule\ condition\ graph$$

4. $\Psi\ (\beta)$: This is the product of $\prod_\sigma\ (\beta)$ and $\prod_\bowtie\ (\beta)$ for a $\beta$ node. This is the selectivity

associated with a $\beta$ node.

$$\Psi\ (\beta) = \prod_{\sigma}\ (\beta) \times \prod_{\bowtie}\ (\beta)$$

The cardinality of a $\beta$ node is estimated as the number of tuples in the cartesian product of the relations corresponding to the $\alpha$ nodes of the subnetwork rooted at the $\beta$ node times the selectivity of all selections and joins associated with the $\beta$ node.

The formula for the cardinality of a $\beta$ node is:

$$\begin{aligned} Card(\beta) &= \prod_{\sigma}\ (\beta) \times \prod_{\bowtie}\ (\beta) \times \prod_{S}\ (\beta) \\ &= \Psi\ (\beta) \times \prod_{S}\ (\beta) \end{aligned}$$

The insertion frequency $F_i(\beta)$ of a $\beta$ node is an estimation of the relative frequency with which tuples would be inserted into this $\beta$ node compared with insertion into other $\alpha$ and $\beta$ nodes. Inserts to a $\beta$ node occur as a result of inserts to $\alpha$ nodes that are leaves of the $\beta$ node. In the following formulae, $R_k$ is $R(\alpha_k)$. The estimated number of tokens inserted into a $\beta$ node when a token $t$ arrives at one of its leaf $\alpha$ nodes is the estimated number of tuples in the join of $t$ with the other leaf $\alpha$ nodes of the $\beta$ node. $F_i(\beta)$ is the sum over the leaves $\alpha_k$ of $\beta$ of $F_i(R_k)$ times the number of tokens generated by a token arrival at $\alpha_k$. Thus, we have:

$$\begin{aligned} F_i(\beta) &= \sum_{\alpha_k \in leaves\ (\beta)} F_i(R_k) \times \Psi\ (\beta) \times \prod_{S}\ (\beta) \times \frac{1}{S(R_k)} \\ &= \sum_{\alpha_k \in leaves\ (\beta)} F_i(R_k) \times Card(\beta) \times \frac{1}{S(R_k)} \end{aligned}$$

The deletion frequency $F_i(\beta)$ of a $\beta$ node is an estimation of the relative number of tuples that would be deleted from the $\beta$ node compared with deletion from other $\alpha$ and $\beta$ nodes.

$$\begin{aligned} F_d(\beta) &= \sum_{\alpha_k \in leaves\ (\beta)} F_d(R_k) \times \Psi\ (\beta) \times \prod_{S}\ (\beta) \times \frac{1}{S(R_k)} \\ &= \sum_{\alpha_k \in leaves\ (\beta)} F_d(R_k) \times Card(\beta) \times \frac{1}{S(R_k)} \end{aligned}$$

The cost of every node in the Gator network is defined recursively. The cost of a node in the Gator network depends on the shape of the subnetwork rooted at that node. In specific, the cost of a $\beta$ node is dependent on the following factors:

- The cost of the children of the $\beta$ node.
- The cost of performing joins for tokens fed into all the children of the $\beta$ node.
- The cost associated with maintaining/updating the $\beta$ node.

The cost of a $\beta$ node can thus be expressed as:

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

The LocalCost($\beta$) of a $\beta$ node is defined by the following equation:

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

The two terms PerChildInsCost(N,$\beta$) and PerChildDelCost(N,$\beta$) indicate the respective costs of processing a "+" and "−" token arriving at a child $N$ of the $\beta$ node. For developing the cost estimates, it is assumed that the tuples arriving at a node are processed one at a time. The method JoinSizeAndCost(N,$\beta$) returns the expected cardinality (result size) and cost for the multi-way join performed when a token arrives at a child $N$ of the $\beta$ node. The method updateCost($\beta$,size) represents the cost of updating the $\beta$ node with *size* number of tuples.

$PerChildInsCost(N,\beta)\{$
$\quad (size, cost) = JoinSizeAndCost(N,\beta)$
$\quad return(cost + updateCost(\beta, size))$
$\}$

Every node $N$ has a join order plan associated with it (how this is determined will be discussed later). When a token arrives at a child $N$ of a $\beta$ node, a sequence of two-way joins based on the join order plan of the node $N$ is performed. This process goes through the following steps:

- Obtain a Temporary Join Result (TR) by joining the incoming token with the contents of the first node in the join order plan of $N$.
- While the TR is not empty and there is a next element in the join order plan, join TR with the contents of the next node in the join order plan.

In the current version of Ariel we use nested loop joins for the above process and TR always forms the outer relation in the nested loop join. At every nested loop join performed, the CPU cost equals the number of tuple comparisons given by the product of the cardinalities of the TR and the inner node. If the inner node fits in one page, then we need only a single I/O. Otherwise, assuming that for every tuple in TR, all the pages of the inner node are read, we would be doing min(Pages(inner node),Card(TR)) number of I/Os. Based on these considerations, the evaluation method JoinSizeAndCost is developed. Nested loop join is used since it is expected TR will normally be small. If an index is present on the join attribute of the memory nodes to which the TR is being joined, that index could be used for the inner scan of the nested loop join. We plan to implement memory node indexes in a future version.

For evaluating the update cost we use the Yao approximation for the number of pages that would be touched when k tuples are randomly searched within relations that occupy m pages [22]. Insertion of a total of *size* number of tuples involves bringing in the required pages and writing them back after updating them. If the incoming tuples occupy more than a page, that many new pages have to be fetched, modified and written back to the disk.

For propagating a delete token we use the technique of delete optimization. Here instead of performing the join as in token insertion, the contents of the $\beta$ memory are scanned for the presence of the tuples having components that match the delete token. In order to do this, we need to scan all the pages of the $\beta$ memory and then have to write back those pages of the $\beta$ node that have components that match the delete token input.

The definitions of the methods JoinSizeAndCost, updateCost and PerChildDelCost follow:

$JoinSizeAndCost(N, \beta)$
{
TRSize = 1
tempCost = 0
for each node $n$ in the join order plan of N
{ if Pages($n$) $\leq 1$
$tempCost =$
$tempCost + I/O_{weight} + CPU_{weight} \times S(n) \times TRSize$
else
tempCost = tempCost + $I/O_{weight} \times min(pages(n), TRSize) +$
$CPU_{weight} \times tuplePerPage(n) \times$ TRSize
TRSize = TRSize $\times S(n) \times JSF(N, n)$
}
return $(TRSize, tempCost)$
}

$updateCost(\beta, size)$
{
$tempCost = Yao(pages(\beta), size) \times 2 \times I/O_{weight} + size \times CPU_{weight}$
if $\lceil \frac{size}{tuplesPerPage(\beta)} \rceil > 1$ then
$tempCost = tempCost + \lceil \frac{size}{tuplesPerPage(\beta)} \rceil \times 2 \times I/O_{weight}$
return($tempCost$)
}

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

### 3.3 Cost Functions for P-node

In our Gator network implementation, the P-node is directly connected to a dedicated transparent-$\beta$ node (trans-$\beta$ for short) that is the root $\beta$ node for a rule. The trans-$\beta$ is just a place holder. It passes its input to the associated P-node directly. The cost functions associated with the P-node are slightly different. The cost function for a P-node is defined as,

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

The method LocalCost(P) is similar in form to the corresponding method for $\beta$ nodes.

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

The P-nodes do not outlive a transaction. As the contents of the P-nodes are emptied whenever the corresponding rules execute, they are normally small enough to be in main memory. This implies that there would be no disk I/O involved while updating a P-node. The small size of a P-node coupled with delete optimization permit us to estimate that the cost associated with deleting a tuple from a P-node as $CPU_{weight}$. The cost associated with deleting a tuple from a P-node is equal to the cost of scanning the P-node in memory.

The cost associated with the insertion or deletion of a tuple from a P-node can be evaluated by the methods $PerChildInsCost(N, P)$ and $PerChildDelCost(N, P)$ defined as shown:

$PerChildInsCost(N, P)$
{
    $(size, cost) = JoinSizeAndCost(N, \beta)$
    return $(cost + CPU_{weight} \times size)$
}

$PerChildDelCost(N, P)$
{
    return $(CPU_{weight})$
}

### 3.4 Evaluation of Selectivity Factors

The selectivity factors Sel($\alpha$) and JSF($N_i$,$N_j$) play a key role in the estimation of the size, frequency and size of the nodes in the Gator discrimination network. These selectivity factors are derived from the statistics maintained by the system using techniques similar to the ones used in query optimizers [18]. In fact, the same selectivity estimators used by the Ariel query optimizer are used by the Gator network optimizer.

## 4 Randomized Optimization Strategy

For a given rule there could many possible Gator networks. The efficiency of the rule condition testing mechanism depends on the shape of the Gator network. We implemented an optimizer in Ariel that uses a randomized state-space search technique to get optimally shaped Gator networks. The optimizer is capable of using the following randomized state-space search strategies: 1. iterative improvement (II), 2. simulated annealing (SA), and 3. two-phase optimization (2PO, a combination of II and SA). The optimizer uses the statistics such as update frequencies, sizes etc., maintained in the catalogs for the optimization process. We were motivated to use a randomized approach to Gator network optimization since it has been used successfully for optimizing large join queries [9], a problem with a similarly large search space. We conducted experiments [8, 16] which demonstrated that a randomized approach is superior to a dynamic programming approach like that used in traditional query optimizers [18]. A general description of II, SA and 2PO is given below, followed by a description of how we apply them to Gator optimization.

### 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 keeps accepting 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.

In Ariel, we implemented a random Gator network constructor that builds a complete feasible Gator network for a rule by making random choices about the Gator network structure, avoiding formation of $\beta$ nodes that are cross-products [14]. This constructor was used to produce random start states for II.

### 4.2 Simulated Annealing

Simulated Annealing (SA) is a Monte Carlo optimization technique proposed by Kirkpatrick et al. [13] for problems with many degrees freedom. This 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. See [13, 11] for more details on SA.

### 4.3 Two Phase Optimization

As the name 'Two Phase Optimization' (2PO) suggests, this approach runs in two phases. In the first phase it 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 [9].

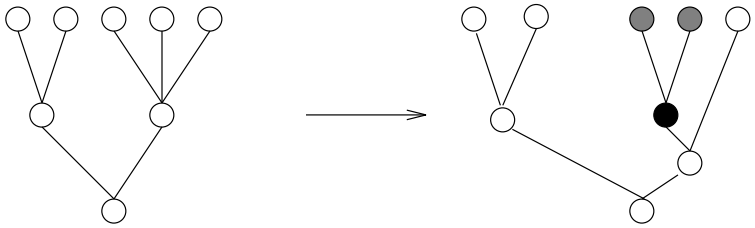
These generic algorithms require the specification of three problem-specific parameters, namely state space, neighbors function and cost function [11, 9, 10]. For the optimization of the Gator network we defined these parameters as below:

1. State Space: The state space of the Gator network optimization problem for a given rule is defined as the set of all possible shapes of the complete Gator network for that rule. Each possible shape of the Gator network corresponds to a state in the state space.

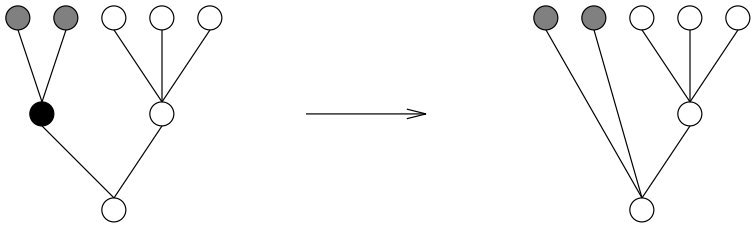
2. Neighbors Function: The neighbors function in the optimization problem is specified by the following set of transformation rules, also shown in figure 6. In the following discussion two sibling nodes in the discrimination network are said to be *connected* if the following holds. First, we define the *condition graph node set* of a Gator network node N, CGNS(N), to be the set of condition graph nodes corresponding the the leaf $\alpha$ 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).

    - *Kill-Beta*: Kill-Beta removes a randomly picked $\beta$ node, say kill-$\beta$, and adds the children of the node kill-$\beta$ as children of the parent of the node kill-$\beta$.

CREATE BETA



KILL BETA



MERGE SIBLING

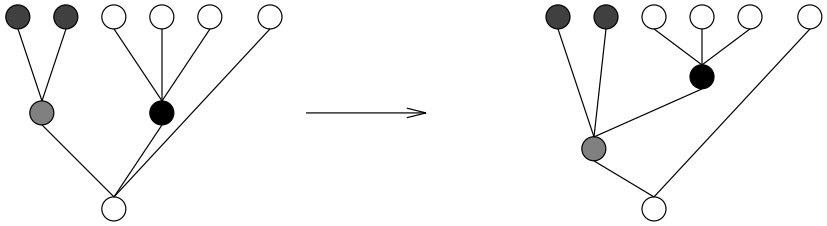


Figure 6: Local change operators

- *Create-Beta*: Create-Beta adds a new $\beta$ node, say create-$\beta$, to the discrimination network. It picks a random $\beta$ node and if this $\beta$ node has more than two children, Create-Beta randomly picks two connected siblings rooted at this $\beta$ node and makes them the children of create-$\beta$. If there is more than one remaining sibling of the original parent $\beta$ node, those siblings must be connected.
- *Merge-Sibling*: Merge-Sibling merges a node into its sibling. It picks a random $\beta$ node. If this $\beta$ node has more than two children, then two connected siblings rooted at this $\beta$ are randomly picked and one of them is made a child of the other. The node to which a child is added must be a $\beta$ .

3. Cost Function: The cost function used is the one given in section 3.

# 5 Modifications to Ariel

The first implementation of the Ariel active DBMS was based on the A-TREAT algorithm. As this was a modification of the TREAT algorithm, it had no $\beta$ nodes. The implementation of the Gator discrimination network for rule condition testing involved several major tasks such as modifications to the existing class hierarchy of the discrimination network node types to include $\beta$ nodes, mechanisms to perform priming the Gator network, generating token join order plans for the nodes in the discrimination network, and mechanisms for propagating the tokens through the network for pattern matching.

## 5.1 New Discrimination Network Node Types

In the original Ariel system, since a variation of TREAT was used, there were no $\beta$ nodes, only $\alpha$ nodes and P-nodes. There were seven different types of $\alpha$ nodes with slightly different behavior (see [4] for details). These were implemented as a hierarchy of classes in the E language [17], a form of persistent C++. 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 $\beta$ nodes:

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

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

## 5.2 Priming the Gator Network

Priming is a process in which the nodes of the discrimination network are loaded with the tuples that match the selection/join predicates of the rule condition subgraphs associated with these nodes. In the Ariel active DBMS the presence of dynamic nodes, i.e dynamic-$\alpha$s and dynamic-$\beta$s, makes this process more complicated.

The contents of the dynamic-$\alpha$s do not outlive a transaction. This implies that the contents of the dynamic-$\beta$s too should be flushed at the end of every transaction. *So while priming the network we need to materialize only those nodes that are not dynamic.* Also during priming, the virtual-$\alpha$s are not materialized.

To prime a stored-$\alpha$, a one-tuple-variable query is formed to retrieve the data to be stored in the $\alpha$ node. This query is formed internally as a query graph by copying the node of the rule condition graph corresponding to the $\alpha$ node. This one-variable query graph is passed to the query optimizer, and the resulting plan is executed. The data retrieved are stored in the $\alpha$ memory. This approach already existed in the earlier implementation.

For priming a $\beta$ node, a nested-loop join plan for joining the children of the $\beta$ node is built using the same logic as in the generation of the node's join order plan (described in section 5.3). This nested-loop join plan is executed and the results are used to prime the $\beta$ node.

Several alternatives to this strategy are possible, as follows:

1. Build a rule-subgraph based on the join edges between the *leaves* of the $\beta$ node. Using the rule-subgraph as an input to the query optimizer, get an optimized plan for priming the $\beta$ nodes. This involves some redundant work to re-evaluate the contents of the $\alpha$ nodes.

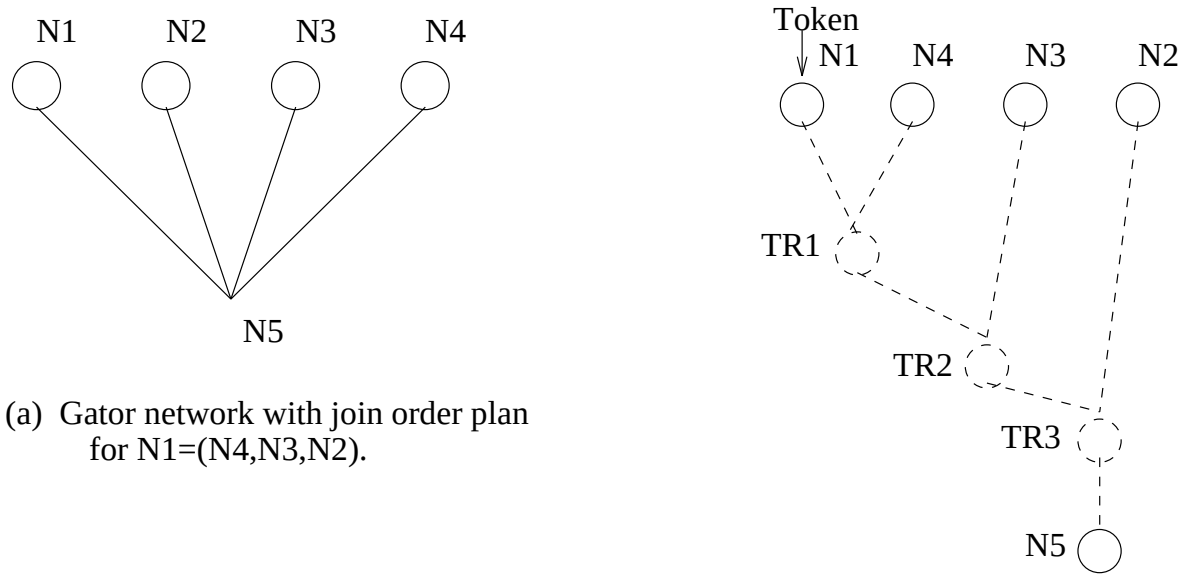
2. Make the system treat $\alpha$ and $\beta$ nodes as relations. Build a rule-subgraph based on the join edges between the *children* of a $\beta$ node. Using the rule-subgraph as an input to the query optimizer, get an optimized plan to prime the $\beta$ node. This approach avoids the redundant work associated with the previous strategy.

Approach 2 is the best approach, and approach 1 is better than the nested-loop join approach we implemented. We chose the nested-loop join approach because it was much simpler to implement than the other two.

The priming algorithm used now is recursive in nature. The Gator network gets primed once it is optimized and made persistent. The priming is done using a post-processor which descends the network from the root towards the leaves. To prime the discrimination network the primeMe method is invoked on the root of the Gator network as in:

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

For stored-$\alpha$ nodes, the method *materializeTuples* realizes the tuples by applying the selection predicate to the associated relation, using the plan produced by the query optimizer. For static-$\beta$ nodes, the method *materializeTuples* realizes the tuples using a nested-loop join between the




(a) Gator network with join order plan for N1=(N4,N3,N2).

(b) Propagating a token arriving at node N1.

Figure 7: Join order plan for a Gator node

children of the static-$\beta$ node. For the dynamic/virtual nodes, *materializeTuples* does not realize any tuples.

### 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 7(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 7(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 rule, 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 very fast.

## 6 Performance Evaluation

Experiments were conducted to evaluate the performance of the Gator discrimination network compared to Rete and TREAT networks. The various metrics used for performance evaluation include optimization time, priming time and token propagation time.

Experiments were done by taking a realistic database and the rule IrisRule shown in section 2. The rule condition graph for the IrisRule is shown in figure 1. The database size, relation cardinalities and the expected update frequencies of the relations are shown in table in figure 8.

| Relation | No.of Tuples | Tuple Size in bytes | Insert Frequency | Delete Frequency |
|---|---|---|---|---|
| customer | 600 | 43 | 0.002 | 0.002 |
| salesperson | 15 | 19 | 0.002 | 0.002 |
| neighborhood | 30 | 19 | 0.002 | 0.002 |
| desired_nh | 600 | 8 | 0.002 | 0.002 |
| covers_nh | 15 | 8 | 0.002 | 0.002 |
| house | 15000 | 56 | 0.990 | 0.990 |

Figure 8: Database Statistics

To measure the optimization time, we let the optimizer run without time constraint to see how the quality of the optimal Gator network (in terms of estimated cost) changes with time. The results of the two optimization algorithms Iterative Improvement (II) and Two-Phase Optimization (2PO) are shown in figures 9 and 10. In figure 9, the "No. of Iterations X 200" on the $x$ axis gives the number of local optimizations. In figure 10, "No. of Iterations X 200" gives the number of local optimizations by II prior to running SA in 2PO. The estimated quality of the Gator network output by II improves by more than a factor of 10 after 600 iterations and it stabilizes after that. II reaches what is apparently the global minimum-cost state after roughly 12 seconds (1000 iterations) for this rule.

II more quickly gets away from very high-cost states and produces a result whose cost never increases as the number of iterations increases. For 600 iterations, 2PO actually found the apparent global minimum state, in fewer iterations than it took II to find it (but around the same amount of time). After 1000 iterations, the SA phase of 2PO always starts with the apparent global minimum cost state, since that is what the II phase finds. Sometimes, the SA phase "pops up" to a higher cost state and then does not find its way back down to the apparent minimum cost state. This is the source of the jagged nature of the curve in figure 10.

In extensive prior simulation results [8, 16] we noticed that in some cases the SA phase did improve the performance, i.e. 2PO produced better output than II. More experiments need to be conducted to understand the shape of the search space and to decide the superiority of one algorithm over another for this problem using our actual implementation. However, we feel it is safe to conclude that an II algorithm is a satisfactory approach to Gator network optimization because it produces good results even for large rule condition graphs, and it is simple to implement, requiring little or no "tuning" to make it work well. 2PO may be better than II in some situations, but 2PO requires careful tuning to make it work well.

Our Gator optimizer can produce optimized Gator networks in a few tens of seconds or less for most conceivable rules. This expense will easily pay for itself by saving time during rule condition matching. The current optimizer implementation, written in E, is not highly tuned. Careful re-implementation could speed it up substantially.

From the experiments above, the best Gator network found is shown in figure 11. The shape of the best Gator network in figure 11 can be explained as follows. For relations customer, covers_nh, desired_nh and house with no selection predicate in the rule condition, virtual $\alpha$ nodes are created preventing the duplication of relations and thus saving space. The $\alpha$ nodes with high update frequency (the $\alpha$ node corresponding to the relation "house") are pushed down the discrimination network towards the root or the P_node of the network. This is logical since this means fewer token

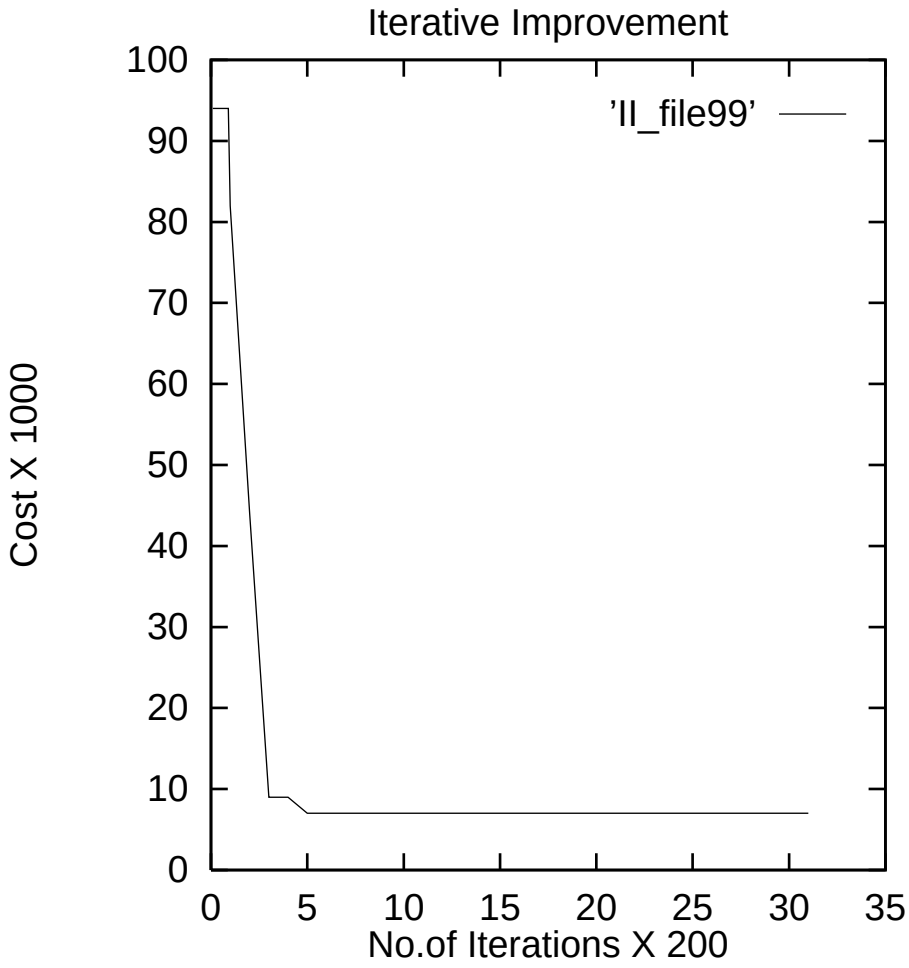



Figure 9: II Results

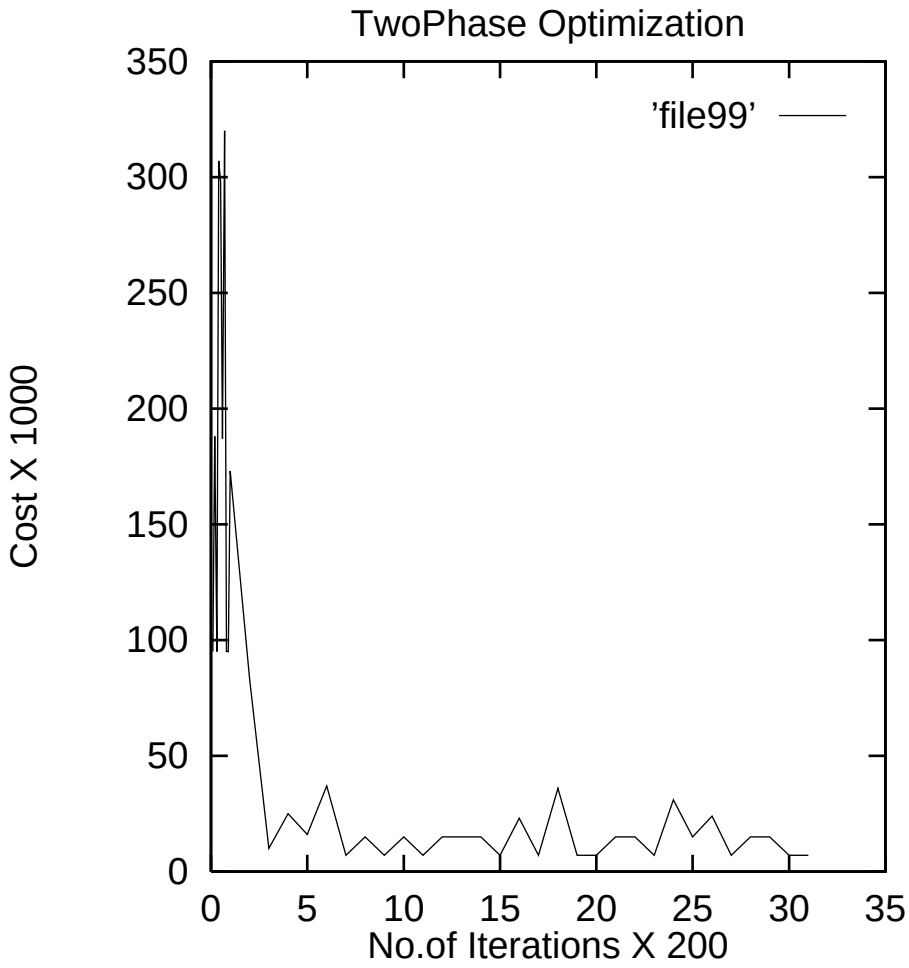



Figure 10: 2PO Results

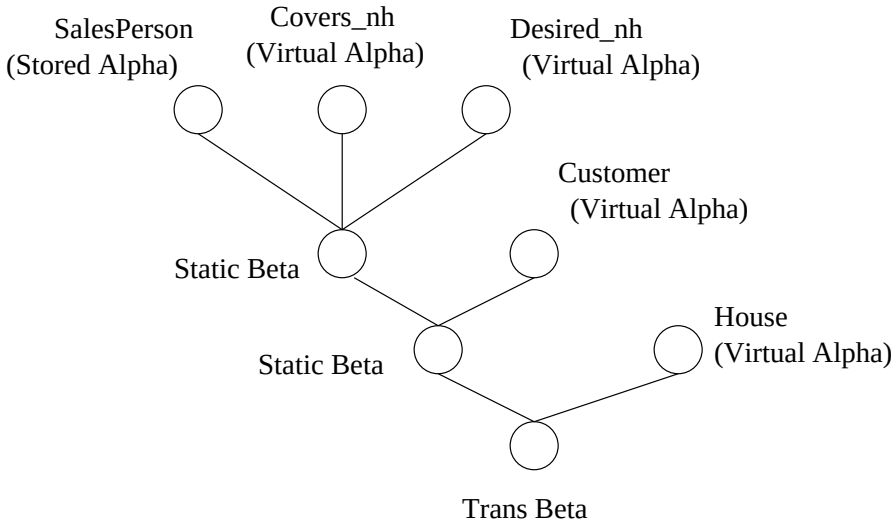



Figure 11: Best Gator Network

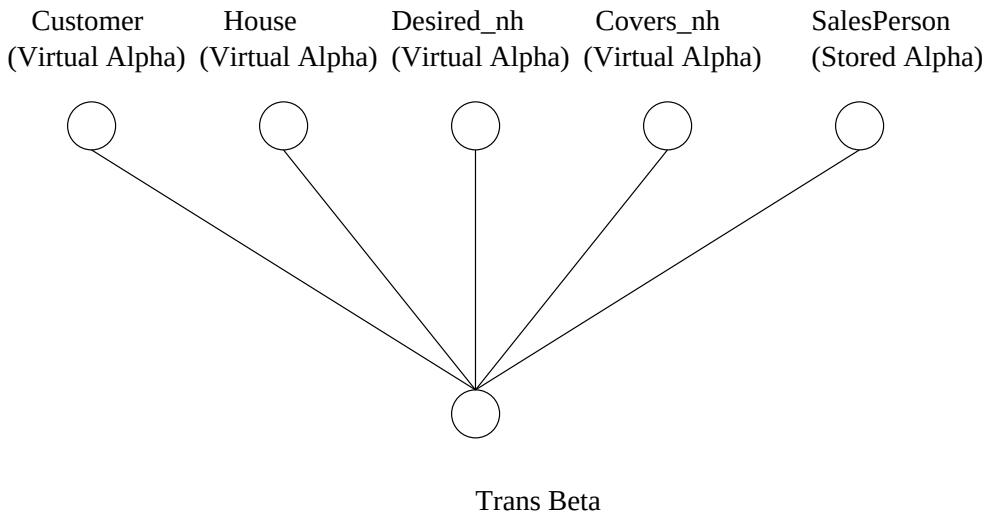



Figure 12: TREAT Network

joins and $\beta$ node updates need to be done as tokens propagate through the network due to updates. This results in improved rule condition testing performance. We can also observe that $\alpha$ Nodes with large size are also pushed down the network. This tends to reduce the size of intermediate $\beta$ nodes and hence results in reducing the amount of state information that needs to be maintained. Another observation is that the $\alpha$ node corresponding to the relation "SalesPerson" is at the top and it is forming a $\beta$ node with two other relations. This is the $\alpha$ node with the highest selectivity and the lowest size and hence it helps to reduce the size of the $\beta$ nodes below it.

The TREAT network for the above rule is shown in figure 12. Here, whenever a new token enters the network, it always has to participate in the join operation with four other $\alpha$ nodes and that explains its higher cost.

The Rete network for the IrisRule is shown in figure 13. This Rete network is not optimized, but it does have the SalesPerson node at the top, which keeps its cost relatively low. The cost could be much worse, e.g., if the large, high-updated-frequency House node was placed at the top and the SalesPerson node was placed at the bottom.

Priming times for the three networks are shown in the table in figure 14. TREAT is the lowest of all followed by Gator and Rete. While TREAT does not maintain any join state information ($\beta$ nodes), Gator maintains minimal state information followed by Rete. This explains the higher Gator and Rete priming times. Priming times are relatively slow, indicating that the simple nested loop join strategy for materializing $\beta$ nodes is not satisfactory. In a later version of the system, we plan to switch to using optimized query plans for $\beta$ node priming. These plans will be able to use sort-merge join in addition to nested loop join. This will dramatically speed up priming time.

Changes to the database are represented by tokens that are passed to the discrimination network. Average token propagation time gives an estimate of the overhead due to rule condition testing while processing updates. We measured the average token propagation time by processing 100 updates to the system. The table to which an update was applied was determined using a frequency distribution determined by the update frequency statistics maintained in the system catalog. The deletes are done at the same rate as the inserts so that the database size remains roughly constant. The propagation time is measured for each of the tokens and the average is computed by taking into account their update frequencies. The results are shown in the table in figure 15.

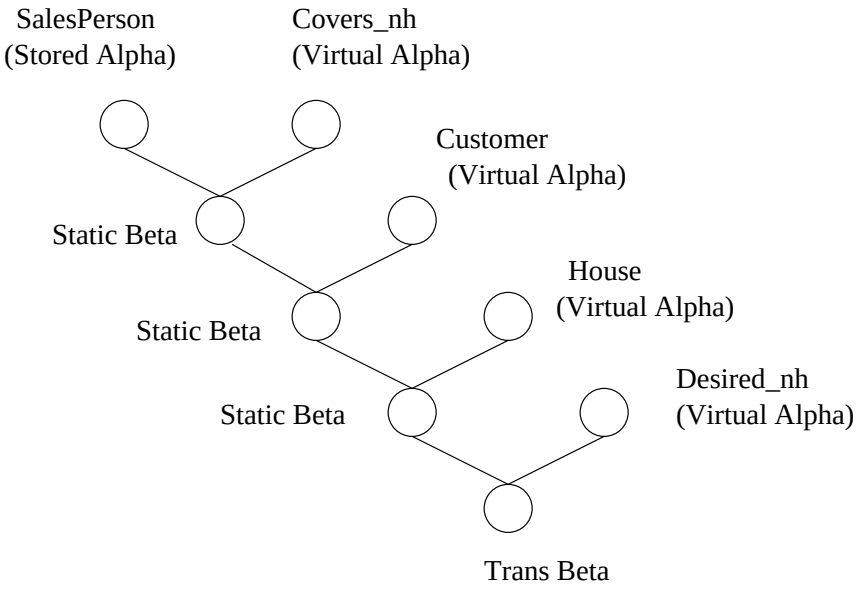



Figure 13: Rete Network

| **Rete** | **BestGator** | **TREAT** |
|---|---|---|
| 426.9 | 338.32 | 280.01 |

Figure 14: Priming Time (in seconds)

| | **Rete** | **TREAT** | **BestGator** |
|---|---|---|---|
| Average Token Propagation Time | 0.60 | 1.13 | 0.34 |
| BestGator SpeedUp | 1.70 | 3.32 | - |

Figure 15: Average Token Propagation Time in seconds

We can see that the optimized Gator network is superior to both Rete and TREAT for IrisRule. The speed up is 3.3 times compared to TREAT and 1.7 times compared to Rete. TREAT does not maintain any state information, and hence any token coming into the system has to do a join with all the other Alpha nodes. The Gator network is optimized for the given cardinalities and update frequencies, which makes it the fastest of all. An optimized Rete network has the potential to perform close to the best Gator network for this database configuration (with skewed update frequencies). A poorly chosen Rete network could perform much worse then TREAT. Since Gator can be optimized for any database and update frequency configuration, it has the potential to outperform Rete and TREAT in all cases. More experiments are needed to more fully characterize the behavior of Gator. Future work is also needed regarding modification or re-optimization of Gator networks as database size and update statistics change.

# 7 Conclusion

We have introduced Gator networks, a new discrimination network structure for optimized rule condition testing in active databases. A cost model for Gator has been presented, 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. Iterative improvement has been shown to be a satisfactory optimization strategy for Gator networks. Further work is needed to see whether it is worthwhile to use the more complex two-phase optimization scheme. Randomized optimization of Gator networks is crucial. Our previous simulation results [8, 16] illustrate that a traditional dynamic programming approach to Gator network optimization takes too long to optimize rules with more than seven tuple variables. Moreover, a randomized optimizer produces better results (lower-cost Gator networks) even for rules with fewer than seven tuple variables.

Our Gator network cost estimator is adequate in the sense that when using the estimator, the optimizer produces qualitatively "good" Gator networks. A detailed analysis of the cost functions to see how closely they predict the actual cost of the Gator networks generated would be beneficial.

This work has clearly demonstrated the value of optimizing the testing of complex trigger conditions involving joins in active databases. This will help make it feasible to implement the capability of processing triggers with joins in their conditions in commercial database systems, 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. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, June 1992.

[5] Eric N. Hanson. Gator: A discrimination network suitable for optimizing production rule matching. Technical Report TR-007-93, University of Florida CIS Dept., February 1993. http://www.cis.ufl.edu/cis/tech-reports/.

[6] Eric N. Hanson. The design and implementation of the Ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, 1995. (to appear).

[7] Eric N. Hanson, I-Cheng Chen, Roxana Dastur, Kurt Engel, Vijay Ramaswamy, and Chun Xu. Flexible and recoverable interaction between applications and active databases. Technical Report 94-033, University of Florida CIS Department, September 1994. `http://www.cis.ufl.edu/cis/tech-reports/`.

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

[9] 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.

[10] 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.

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

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

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

[14] 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/.

[15] 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.

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

[17] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15(3), 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.

[22] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4), 1977.