The Ariel project has culminated in several advancements in active database technology, and the development of a working prototype active database system. Ariel is unique in its support for efficient rule condition testing based on a discrimination network, and its tight integration of rule processing with database transaction processing. An efficient index for testing single-relation selection predicates was developed, which also resulted in the development of two new types of interval index data structures, the interval binary search tree, and the interval skip-list. For testing join conditions, a modified version of the TREAT algorithm, called A-TREAT, was developed. A-TREAT is suitable for use in a database environment. It’s major new feature that makes it suitable for databases is the concept of virtual alpha-memory nodes, which, unlike normal alpha-memories, do not contain the data matching the associated selection predicate. Instead, virtual alpha-memories contain only the predicate.
The Ariel project has culminated in several advancements in active database technology, and the development of a working prototype active database system. Ariel is unique in its support for efficient rule condition testing based on a discrimination network, and its tight integration of rule processing with database transaction processing. An efficient index for testing single-relation selection predicates was developed, which also resulted in the development of two new types of interval index data structures, the interval binary search tree, and the interval skip-list. For testing join conditions, a modified version of the TREAT algorithm, called A-TREAT, was developed. A-TREAT is suitable for use in a database environment. It’s major new feature that makes it suitable for databases is the concept of virtual α-memory nodes, which, unlike normal α-memories, do not contain the data matching the associated selection predicate. Instead, virtual α-memories contain only the predicate. The contents of a virtual α-memory are materialized on demand. A serious problem with using discrimination networks such as Rete and TREAT developed for artificial intelligence programming systems in active databases is that the memory nodes require too much storage. The virtual memory nodes of A-TREAT eliminate this problem. Other unique feature of Ariel are its uniform treatment of rules with conditions based on patterns, events, and transitions, and its efficient execution of rule actions using optimal plans generated by the rule action planner. The Ariel prototype is implemented on top of the EXODUS toolkit using the E programming language, a version of C++ extended with persistent objects. The prototype supports queries, updates and rule processing.
1 Introduction

This report summarizes the outcome of the Ariel project, a research project into active database management funded by grant AFOSR-89-0286. A detailed description of the Ariel system can be found in [12, 8, 10]. A complete list of publications related to the Ariel project includes [7, 12, 26, 8, 10, 11, 9, 13, 25, 4, 3, 14, 28, 18, 17]. Database researchers and practitioners have become quite interested in active database systems recently. The source of this interest is a strong application need. Many applications in areas such as C^3I, medical informatics, financial data management, and general business data processing are filled with requirements for alerting users or otherwise taking action based on changes to the database. Active database research is directly targeted at meeting these needs. Of particular interest to the Air Force is how active database management systems (DBMSs) can support C^3I applications. The Air Force also can benefit from use of active databases to support its many other activities that depend on database systems.

Ariel is a relational database system that has been augmented to support forward-chaining rules with the following general form:

```
define rule rule-name [in ruleset-name]
[priority priority-val]
[on event]
[if condition]
then action
```

A unique rule-name is required for each rule so the rule can be referred to later by the user. The user can optionally specify a ruleset name to place the rule in a ruleset. Rulesets are simply a means of grouping rules together for programmer convenience. The on clause can specify an event-based condition, and the if clause can specify a pattern-based condition. The optional priority clause can be used to specify relative priority between rules to allow the system to decide which rule to execute first in the case that multiple rules are eligible to run. The rule action can contain one or more database commands.

Consider the following example schema from a logistics database:

```
depots(dno, name, location)
supplies(dno, pno, quantity_on_hand)
parts(pno, category, name, description)
orders(pno, dno, quantity)
```

An example rule based on this schema is:

```
define rule reorderBolt
if depots.name="Defense Electronic Supply Center"
and depots.dno = supplies.dno
and supplies.quantity_on_hand \leq 30
and previous supplies.quantity_on_hand > 30
and supplies.pno = parts.pno
and parts.category = "bolt"
then append to orders(parts.pno, depots.dno, 50)
```

The meaning of this rule is to order 50 more of any type of bolt for the Defense Electronic Supply Center if the quantity on hand there drops to 30 or less.
The Ariel project has generated several new techniques for use in active DBMSs, as well as a working prototype active DBMS. Section 2 of this report describes the overall architecture of Ariel. Section 3 describes the new discrimination network developed for Ariel which is tailored for high performance in an active DBMS. Section 4 discusses how Ariel uniformly handles rules with pattern-based, event-based, and transition conditions. Section 5 covers how Ariel generates optimized plans for rule actions, and how those plans are executed. The Ariel prototype is described in section 6. Finally, section 7 discusses related work, and section 8 presents conclusions.

2 Architecture

The architecture of Ariel, shown in Figure 1, is similar to that of System R [1] with additional components attached for rule processing. When commands enter Ariel they are processed by the lexer, parser, and semantic analyzer. If they are queries or data manipulation commands, they are passed to the query optimizer. Execution plans produced by the optimizer are carried out by the query plan executor. The executor is built on top of the storage system provided by the EXODUS database toolkit [2, 15]. In addition to the standard components, Ariel has a rule manager to handle creation and activation of rules, a rule catalog for maintaining the definitions of rules, a discrimination network for testing rule conditions, a rule execution monitor for carrying out rule execution, and a rule action planner for binding the data matching a rule condition with the rule action and producing an execution plan for that action.

3 Discrimination Network

An efficient strategy for incrementally testing rule conditions as small changes in the database occur is critical for fast rule processing. Ariel contains a rule condition testing network called A-TREAT (short for Ariel TREAT) which is designed to both speed up rule processing in a database environment and reduce storage requirements compared with TREAT. An important performance optimization in A-TREAT is the use of a special top-level discrimination network for testing single-relation selection conditions of rules [12]. In addition, we introduce a technique for reducing the
amount of state information stored in the network, whereby \( \alpha \)-memory nodes are replaced in some cases by virtual \( \alpha \)-memory nodes which contain only the predicate associated with the node, not the tuples matching the predicate. In addition to these performance enhancement techniques, we have developed some enhancements to the standard TREAT network in order to effectively test both transition and event-based conditions with a minimum of restrictions on how such conditions can be used.

3.1 Selection Network
Ariel uses a special index optimized for testing selection conditions as the top layer in its discrimination network. This index makes use of an interval index called the interval binary search tree to efficiently test conditions that specify closed intervals (e.g., constant1 < relation.attribute \( \leq \) constant2), open intervals (e.g., constant < relation.attribute), or points (e.g., constant = relation.attribute). Readers are referred to [12] for a detailed discussion of the selection condition testing network in Ariel. A data structure called the interval skip list [9] can be used as an interval index in place of the interval binary search tree discussed in [12, 11]. The interval skip list is much easier to implement than the IBS tree and performs as well.

3.2 Join Network
Here we describe a variation of the Rete and TREAT algorithms for minimizing storage use in database rule systems. In the standard Rete and TREAT algorithms, there is an \( \alpha \)-memory node for every selection condition on every tuple-variable present in a rule condition. If the selection conditions are highly selective, this is not a problem since the \( \alpha \)-memories will be small. However, if selection conditions have low selectivity, then a large fraction of the tuples in the database will qualify, and \( \alpha \)-memories will contain a large amount of data that is redundant since it is already stored in base tables. Storing large amounts of duplicate data is not acceptable in a database environment since the data tables themselves can be huge.

In order to avoid this problem, for memory nodes that would contain a large amount of data, a virtual memory node can be used which contains a predicate describing the contents of the node rather than the qualifying data itself. In a sense, this virtual node is a database view. When the virtual node is accessed, the (possibly modified) predicate stored in the node is processed to derive the value of the node. The predicate can be modified by substituting constants from a token in place of variables in the predicate to make the predicate more selective and thus reduce processing time.

The algorithm for processing a single insertion token \( t \) in a TREAT network containing a mixture of stored and virtual \( \alpha \)-memory nodes is as follows. A stored \( \alpha \)-memory node contains a collection \( C \) of the tuples matching the associated selection predicate. A virtual \( \alpha \)-memory node contains \( P \) and the identifier of the relation \( R \) on which \( P \) is defined. In addition, each transaction \( T \) maintains a data structure ProcessedMemories containing a set of the identifiers of the virtual \( \alpha \)-memory nodes in which token \( t \) has been inserted. ProcessedMemories is emptied before processing of each token.

Suppose a single tuple \( X \) is to be inserted in \( R \). Before putting \( X \) in \( R \), create a token \( t \) from \( X \) and propagate \( t \) through the selection network. When \( t \) filters through the network to an \( \alpha \)-memory node \( A \), the identifier of \( A \) is placed in ProcessedMemories and then \( t \) is joined to neighboring \( \alpha \)-memories. When joining \( t \) to a memory node \( A' \), if \( A' \) is a normal \( \alpha \)-memory, everything proceeds as in the standard TREAT algorithm. If \( A' \) is virtual, then join \( t \) through to the base relation \( R' \) identified in \( A' \) using predicate \( P' \) of \( A' \) as a filter. In addition, if ProcessedMemories contains \( A' \),
then \( t \) belongs to \( A' \). Hence, we must try to join the copy of \( t \) just placed in \( A \) to the copy of \( t \) in \( A' \). If \( t \) joins to itself, a compound token is created and the process continues. At the end of processing \( t \), empty ProcessedMemories, and then insert tuple \( X \) in \( R \). An analogous procedure is used for processing a deletion (-) token.

The algorithm just described has the same effect as the normal TREAT strategy because at every step, a virtual \( \alpha \)-memory node implicitly contains exactly the same set of tokens as a stored \( \alpha \)-memory node. This ensures that if a token joins to itself, it does so exactly the right number of times. A TREAT-based join condition testing algorithm enhanced with virtual \( \alpha \)-memories has been implemented in the Ariel system.

This database schema will be used in some examples to follow:

\[
\begin{align*}
\text{emp(name, age, salary, dno, jno)} \\
\text{dept(dno, name, building)} \\
\text{job(jno, title, paygrade, description)}
\end{align*}
\]

The following rule will be used to illustrate a standard TREAT network, and an A-TREAT network that accomplishes the same task:

\[
\begin{align*}
\text{define rule SalesClerkRule} \\
\text{if emp.sal > 30000} \\
\text{and emp.dno = dept.dno} \\
\text{and dept.name = "Sales"} \\
\text{and emp.jno = job.jno} \\
\text{and job.title = "Clerk"} \\
\text{then action}
\end{align*}
\]

The TREAT network for the rule SalesClerkRule is shown in Figure 2. An A-TREAT network for the rule is shown in Figure 3. The A-TREAT network is identical to the TREAT network, except that the middle \( \alpha \)-memory node (alpha2) is virtual, as indicated by the dashed box around it. If the predicate \( \text{sal} > 30000 \) is not very selective, then making alpha2 be virtual may be a reasonable choice for SalesClerkRule since it can save a significant amount of storage.

The ability to use virtual memory nodes opens up several possible avenues of investigation. It allows trading space for time in a Rete or TREAT network. When to use a virtual memory node
and when not to use one is an interesting optimization problem. Also, the base relation scan done when joining a token to a virtual α-memory can be done with any scan algorithm – index scan or sequential scan. Some optimization strategy is needed to decide whether or not to use an index if one is available, depending on the type of index (primary or secondary, hash or B-tree etc.) and the size of the base relation.

4 Uniform Treatment of all Rule types

Quite unlike standard production systems, Ariel allows rules with transition and event-based conditions in addition to normal conditions. To integrate all these types of conditions into a coherent framework, we generalized the notions of both tokens and α-memory nodes.

4.1 Identifying Transitions

To accommodate transitions, in addition to standard + and - tokens, Ariel uses Δ+ and Δ- tokens which contain a (new,old) pair for a tuple with the value it had before and after being updated. A Δ+-token inserts a new transition event into the rule network, and a Δ--token removes a transition event from the rule network. In addition, all tokens have an event-specifier of one of the following forms to indicate the type of event which created the token:

- append
- delete
- replace(target-list)

The target-list included with the replace event specifier indicates which fields of the tuple contained in the token were updated. On-conditions in the top-level discrimination network are the only conditions that ever examine the event-specifier on a token. Tokens with their event-specifier are also called eventTokens.

In order to send the correct type of token through the network at the correct time, Ariel builds a data structure containing a pair of Δ-sets [I,M] for each relation updated during a transition.
Set I contains an entry for each tuple which was inserted during the current transition. Set M contains an entry for each tuple that existed in the relation at the beginning of the transition and was modified during the transition. It is not necessary to maintain a third set for deletions since once a tuple is deleted it cannot be accessed again.

A \( \Delta \)-set (I or M) contains a set of entries with the following contents:

- **eventSpecifier**: one of `append` or `replace(target-list)`, describing the type of event that created the entry.
- **isDelta**: true or false.
- **tupleValue**: a single tuple if isDelta is false, or a pair of old and new tuple values concatenated together if isDelta is true.
- **descriptor**: a pointer to a format descriptor describing the locations of fields in tupleValue.

The possible sequences of operations that may occur to a single tuple during a transition are one of the following:

- insert followed by zero or more modifications,
- insert followed by zero or more modifications followed by delete,
- one or more modifications,
- zero or more modifications, followed by deletion.

Using the I and M sets, Ariel determines how to update the I and M sets and generate the correct type of token to pass through the discrimination network for operation in each of the four possible sequences listed above [14, 10].

4.2 Identifying Event and Transition Conditions

If a tuple variable appears in the `on` clause of an Ariel rule condition, then the selection condition defined on that variable is considered to be an event-based condition. Similarly, if any tuple variable in the condition has a `previous` keyword in front of it, then the selection condition associated with that variable is a transition condition. Both transition and event-based conditions have the property that the data matching them is relevant only during the transition in which the matching occurred. Afterwards, the binding between the matching data and the condition should be broken. This is accomplished in Ariel using \( \alpha \)-memory nodes that are dynamic, i.e., they only retain their contents during the current transition.

4.3 Summary of Token and \( \alpha \)-memory Types

In general, for the Ariel rule condition testing system we have identified four kinds of tokens and seven kinds of \( \alpha \)-memory nodes. The token types are:

- + token for insertion of a new tuple,
- - token for deletion of a tuple,
- \( \Delta+ \) token for insertion of a new transition token (new/old pair),
- \( \Delta- \) token for deletion of an old transition token.
The α-memory node types include:

**stored-α** standard memory node holding a collection of tuples matching the associated selection predicate,

**virtual-α** virtual memory node holding the predicate but not a collection of matching tuples,

**dynamic-ON-α** a dynamic memory node for an ON-condition which has a temporary tuple collection that is flushed after each database transition,

**dynamic-TRANS-α** a dynamic memory node for a transition-condition which is also flushed after each transition.

**simple-α** an alpha memory for a simple selection predicate for a rule with only one tuple variable in its condition. Simple memories are only used when the rule has just one tuple variable in its condition. Simple memories never contain a persistent collection of the data matching the conditions associated with them since matching data is passed directly to the P-nodes.

**simple-TRANS-α** A simple memory node for a transition condition.

**simple-ON-α** A simple memory node for an event-based (ON) condition.

A different action needs to be taken when each type of token arrives at each type of memory node. The actions for each of the possible combinations are shown in the table in Figure 4. In the table, “π<sub>new</sub>t” represents projection of just the new part of the new/old pair contained in t. A “don’t care” entry indicates that the combination can never occur, since normal + and − tokens can never match a transition condition.

The information in this chart allows the standard TREAT or Rete algorithm to be generalized to handle normal pattern-based conditions as well as event-based and transition conditions, changing only the behavior of individual components, not the overall structure or information flow. This strategy is one of the keys to successful use of TREAT to support condition testing for the Ariel rule language.

This concludes the discussion of how rule conditions are tested in Ariel. We now turn to the problem of how to execute a rule action once it has been determined that the rule should fire.
define rule SalesClerkRule2
if emp.sal > 30000
and emp.jno = job.jno
and job.title = "Clerk"
then do
    append to salaryWatch(emp.all)
    replace emp (sal = 30000)
    where emp.dno = dept.dno
    and dept.name = "Sales"
    replace emp (sal = 25000)
    where emp.dno = dept.dno
    and dept.name != "Sales"
end

Figure 5: Example rule to illustrate query modification.

5 Rule Action Planner and Executor

At the time an Ariel rule is scheduled for execution, the data matching the rule condition is stored in the P-node for the rule. Binding between the condition and action of an Ariel rule is indicated by using the same tuple variable in both. These tuple variables are called shared. To run the action of the rule, a query execution plan for each command in the action is generated by the query optimizer. Shared tuple variables implicitly range over the P-node. When a command in the rule action is executed, actual tuples are bound to the shared tuple variables by including a scan of the P-node in the execution plan for the command. Optimization and execution of Ariel rule actions is discussed in detail below, and illustrated using an example.

5.1 Query modification

When an Ariel rule is first defined, its definition, represented as a syntax tree, is placed in the rule catalog. At the time the rule is activated, the discrimination network for the rule is constructed, and the binding between the condition and action of the rule is made explicit through a process of query modification [21], after which the modified definition of the rule is stored in the rule catalog. During query modification, references to tuple variables shared between the rule condition and the rule action are transformed into explicit references to the P-node. Specifically, for a tuple variable V found in both the condition and action, every occurrence of an expression of the form V.attribute is replaced by P.V.attribute. In addition, if V is the target relation of a replace or delete command, then it is replaced by P.V, and the command is modified to be replace' or delete' as appropriate. The commands replace' and delete' behave similarly to the standard replace and delete commands, except that the tuples to be modified or deleted are located by using tuple identifiers that are part of tuples in the P-node, rather than by performing a scan of the relation to be updated.

For example, consider the rule shown in Figure 5. After query modification is performed on this rule, the commands in its action look as shown in Figure 6, where P is a tuple variable that ranges over the P-node. The tuple variable emp which appears both in the condition and action of the rule has been replaced throughout the action by P.emp in Figure 6. Also, the replace and delete
then do
  append to salaryWatch(P.emp.all)
  replace P.emp (sal = 30000)
  where P.emp.dno = dept.dno
  and dept.name = “Sales”
  replace P.emp (sal = 25000)
  where P.emp.dno = dept.dno
  and dept.name != “Sales”
end

Figure 6: Rule action after query modification.

commands have been transformed into replace' and delete', respectively. The tuple variable dept which does not appear in the condition is unchanged in the action.

5.2 Rule action query plan construction

To execute a command in the rule action, an execution plan for that command must be generated, and this plan must include an operator to scan the P-node if any tuple variables in the command also appear in the rule condition. The Ariel query processor provides an operator called PnodeScan which can scan a P-node and optionally apply a selection predicate to it. When the query optimizer sees the special tuple variable P, it always generates a PnodeScan to find tuples to be bound to P. The rest of the query plan is constructed as usual by the query optimizer. For example, consider construction of the plan for the following command from the action of the rule SalesClerkRule2:

replace P.emp (sal = 30000)
  where P.emp.dno = dept.dno
  and dept.name = “Sales”

The data to be updated by this command are identified by running a query plan which scans P and dept, and joins tuples from these scans. The tuple identifier of the emp sub-tuples bound to the variable P is extracted and used to locate the emp tuples to update. One possible query plan that uses a nested loop join, a PnodeScan on P, and an index scan on dept, is shown in Figure 7. The query optimizer is free to choose the best operators for other operations in the plan besides the PnodeScan, e.g., it could have chosen SortMergeJoin instead of NestedLoopJoin in Figure 7.

5.3 Time of Rule Plan Construction

The time a rule action plan is constructed can have a substantial impact on performance. Our implementation uses a strategy called always reoptimize that produces all plans for execution
of rule actions at rule fire time. Other strategies can be developed which attempt to pre-optimize plans for rule actions, store them, and retrieve them at rule fire time to avoid the cost of run-time optimization [8]. All strategies that store plans must maintain the dependencies between those plans and database objects the plans touch such as tables and indexes, which makes those strategies more complicated from the outset. Moreover, pre-planning strategies are all subject to errors where they run non-optimal plans, whereas always recompute always runs the optimal plan. A thorough investigation of pre-planning strategies vs. always recompute is a potential topic for future investigation.

6 Prototype

Ariel is implemented using the EXODUS toolkit [2, 15] and in particular the E programming language [16], an extension of C++ with persistent objects. The current version of Ariel consists of about 28000 lines of C++/E code. Persistent objects simplified implementation of our catalogs and the rule index. The object-oriented programming features of C++ simplified and streamlined our design [13].

7 Related Work

There has been a significant amount of research on active databases recently. The main thing that differentiates Ariel from other active database systems is its use of a discrimination network specially designed for testing rule conditions efficiently. Other database rule system projects either:

- do not address the need for efficient data structures for finding which rules match a particular tuple (RPL [6], Starburst rule system [27]),
- do not provide a data structure for testing selection conditions, or
- provide a data structure for testing selection conditions which cannot efficiently handle conditions placed on an arbitrary attribute (e.g., one without an index) (POSTGRES rule system [23, 22, 24], HiPAC [5], DIPS [20], Alert [19]).

Other distinguishing features of Ariel are its close adherence to the production system model, its unified treatment of rules with normal conditions as well as event-based and transition conditions, its ability to run rule action commands without creating any additional joins to the P-node, and its use of a rule-action planner that produces optimal plans for executing rule actions.

8 Conclusion

The Ariel project has shown that a database system can be built with an active rule system that is: (1) based on the production system model, (2) set-oriented, (3) tightly integrated with the DBMS and (4) implemented in efficiently using (a) a specially designed discrimination network, and (b) a rule-action planner that takes advantage of the existing query optimizer. Ariel is unique in its use of a selection-predicate index that can efficiently test point, interval and range predicates of rules on any attribute of a relation, regardless of whether indexes to support searching (e.g., B+-trees) exist on the attribute. In addition, the concept of virtual α- (and β-) memory nodes introduced in Ariel can save a tremendous amount of storage, yet still allow efficient testing of rules with joins in their conditions. The ability to use virtual memory nodes in a database rule
system discrimination network opens up tremendous possibilities for optimization, in which the most worthy memory nodes would be materialized for the best possible performance given the available storage. Prior to the development of the virtual memory node concept, it was mandatory to materialize the alpha-memory nodes, limiting potential optimizations.

Transformation of databases from passive to active is a landmark in the evolution of DBMS technology. We hope the development of fast, robust active database systems that may come from this research will lead to innovative new applications that make more productive use of the information in the DBMS of the future.

References


