The Design and Implementation of the Ariel Active Database Rule System[†]

Eric N. Hanson
Department of Computer and Information Sciences
University of Florida
Gainesville, FL 32611
hanson@cis.ufl.edu

UF-CIS-018-92

September 1991

Abstract

This paper describes the design and implementation of the Ariel DBMS and it's tightly-coupled forward-chaining rule system. The query language of Ariel is a subset of POSTQUEL, extended with a new production-rule sublanguage. Ariel supports traditional relational database query and update operations efficiently, using a System R-like query processing strategy. In addition, the Ariel rule system is tightly coupled with query and update processing. Ariel rules can have conditions based on a mix of patterns, events, and transitions. For testing rule conditions, Ariel makes use of a discrimination network composed of a special data structure for testing single-relation selection conditions efficiently, and a modified version of the TREAT algorithm, called A-TREAT, for testing join conditions. The key modification to TREAT (which could also be used in the Rete algorithm) is the use of *virtual* α-memory nodes which save storage since they contain only the predicate associated with the memory node instead of copies of data matching the predicate. The rule-action executor in Ariel binds the data matching a rule's condition to the action of the rule at rule fire time, and executes the rule action using the query processor.

1 Introduction

Designers of database management systems have long wanted to transform databases from passive repositories for data into *active* systems that can respond immediately to a change in the state of the data, an event, or a transition between states [BC79, Esw76]. However, to create a successful active database system, many problems must be solved, including:

[†]This work was supported in part by the Air Force Office of Scientific Research under grant number AFOSR-89-0286. This paper also appeared as Wright State University report WSU-CS-91-06. A shorter version with some new material appeared in the Proceedings of the ACM SIGMOD Conference, June 1992.

- design of a suitable language for expressing active rules,
- design of a condition-testing mechanism for rules that is efficient enough to still allow fast transaction processing,
- integration of rule condition testing and execution with the transaction processing system,
- design of a protocol for allowing rule actions to interact with software external to the DBMS.

The Ariel system is an implementation of a relational DBMS with a built in rule system which has been designed to address the above issues. The Ariel rule system (ARS) is based on the production system model [For82]. Our approach has been to adopt as much as possible from previous work on main-memory production systems such as OPS5 [For81], but make changes where necessary to improve the functionality and performance of a production system in a database environment. The features of Ariel that distinguish it from other commercial and research active database rule systems are the following:

- Ariel is a complete implementation of a relational DBMS with a rule system that is *tightly coupled* with the query processor,
- the design of Ariel places strong emphasis on efficient testing of rule conditions in a database environment, and a high-performance discrimination network for testing rule conditions in that environment has been designed and implemented.

Some other database rule systems have been developed but have not been implemented in a tightly coupled fashion with the database query processor. These include DIPS [SLR89, RSL89] and RPL [DE88a, DE88b]. Another, HiPAC [DBB+88, Cha89, MD89], has been implemented, but only as a main-memory prototype. The POSTGRES rule system (PRS) [SHP88, SRH90] and the Starburst rule system (SRS) [WCL91, HCL+90] have been implemented in a tightly-coupled fashion with their respective database systems. However, neither the PRS, SRS, DIPS, RPL, nor HiPAC have a rule condition testing network comparable to the one in Ariel.

This paper describes the design and implementation of the Ariel DBMS with particular emphasis on the ARS. Section 2 describes the query and rule languages used in Ariel. Section 3 gives an overview of the Ariel system architecture. Section 4 discusses the Ariel *rule catalog*. Section 5 describes the *rule execution monitor* which controls execution of triggered rules. Section 6 presents the structure of tokens that are created by database operations, and the discrimination network used in Ariel for efficiently testing both selection and join conditions of rules against those tokens. Section 7 describes optimization and execution of rule actions. Section 8 gives some performance results. Finally, section 10 reviews related research, and section 11 summarizes and presents conclusions.

2 The Ariel Query and Rule Languages

This section describes the Ariel query and rule languages. The syntax of Ariel rule language (ARL) is covered in section 2.2. The Semantics of ARL rule execution is discussed in section 2.3.

2.1 Query Language

The focus of the Ariel project is on the rule system, so we made the decision to use well-understood database technology for the other parts of the system wherever possible. We thus decided to use the relational data model and provide a subset of the POSTQUEL query language of POSTGRES

for specifying data definition commands, queries and updates [SRH90]. POSTQUEL commands **retrieve**, **append**, **delete**, and **replace**, are supported, along with other commands for creating and destroying relations and indexes, and performing utility functions such as loading relations, gathering statistics on data in relations, and so forth. The syntax of POSTQUEL data manipulation commands is shown below. Square brackets indicate optional clauses.

retrieve (*target-list*)
[**from** *from-list*]
[**where** *qualification*]

append [**to**] *target-relation* (*target-list*)
[**from** *from-list*]
[**where** *qualification*]

delete *tuple-variable*
[**from** *from-list*]
[**where** *qualification*]

replace [**to**] *tuple-variable* (*target-list*)
[**from** *from-list*]
[**where** *qualification*]

In POSTQUEL, the *target-list* is used to specify fields to be retrieved or updated, the *from-list* is used to specify tuple variable bindings, and the *qualification* is used to specify a predicate that the data affected by the command must match. In addition, a relation name can be used as a tuple variable name by default, avoiding the need to use a **from** clause in most cases.

Some relations which will be used throughout the paper are the following:

emp(name, age, salary, dno, jno)
dept(dno, name, building)
job(jno, title, paygrade, description)

An example command to retrieve the name and job title of everyone in the "Toy" department is:

retrieve (emp.name, job.title)
where emp.dno = dept.dno
and emp.jno = job.jno
and dept.name = "Toy"

An equivalent command using a tuple variable e to range over the emp relation is:

retrieve (e.name, job.title)
from e **in** emp

where e.dno = dept.dno
and e.jno = job.jno
and dept.name = "Toy"

For a more detailed description of POSTQUEL, readers are referred to [SRH90]. We now turn to a discussion of ARL.

2.2 Rule Language

ARL is a production-rule language with enhancements for defining rules with conditions based not only on patterns, but also on events and transitions. The ARL syntax is based on the syntax of the query language. Hence, the syntax of the pattern in a rule condition is identical to that for the **where** clause of a query. The general form of an ARL rule is the following:

define rule *rule-name* [**in** *ruleset-name*]
[**priority** *priority-val*]
[**on** *event*]
[**from** *from-list*]
[**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 (use of rulesets will be discussed later). If no ruleset name is specified, the rule is placed in the system-defined ruleset *default_rules*. The **priority** clause allows specification of a priority to control the order of rule execution. The **on** clause allows specification of an event that will trigger the rule. The following types of events can be specified after an **on** clause:

- **append** [**to**] *relation-name*
- **delete** [**from**] *relation-name*
- **replace** [**to**] *relation-name* [(*attribute-list*)]

The *condition* after the **if** clause has the following form:

qualification [**from** *from-list*]

The *qualification* part of a rule's **if** condition has the same form as the qualification of a **where** clause in a query, with some exceptions. One exception is that Ariel does not currently support aggregates in rules because testing aggregate rule conditions can be very time-consuming. The benefits of aggregate conditions were not thought to be worth the cost for the initial Ariel prototype.

The **then** part of the rule contains the action to be performed when the rule fires. The action can be a single data manipulation command, or a *compound command* which is a **do** ... **end** block surrounding a list of commands.

The **from** clause is for specifying bindings of tuple variables to relations. Relation names can be used as default tuple variables in both rules and queries.

An **if** *condition* specifies a logical predicate, but no target list. No target list is specified because the relational projection operation is not allowed in rule conditions. The decision not to allow

projection was made since handling projection would require the system to maintain more state information between updates, and would require extra effort to maintain duplicate counts. The usefulness of projection in rule conditions was not felt to be worth the performance disadvantage.

There will be cases where a rule must be awakened when any new tuple value is created in a relation (due to an **append** or a **replace**). Since no target list is allowed in rule conditions, we provide the following conditional expression to reference a relation:

new (*tuple-variable*)

New can be thought of as a selection condition which is always "true."

2.3 Rule Semantics

The Ariel rule system uses a production system model, where the "working memory" is stored in the database relations and rules are stored separately in the *rule catalog*. Execution of rules is governed by a *recognize-act cycle* similar to that used in OPS5 [For82]. Ariel rules get an opportunity to wake up after every database *transition*. Below, we describe in detail Ariel's treatment of transitions, events, the rule execution cycle, and rule priorities.

2.3.1 Transitions

A transition in Ariel is defined to be the changes in the database induced by either a single command, or a **do ... end** block containing a list of simple commands. Blocks may not be nested. The programmer designing a database transaction thus has control over where transitions occur. If desired, the programmer can put a **do ... end** block around all the commands in the transaction so the entire transaction is a single transition. Each command in a transaction will be considered a transition by itself unless it is enclosed in a block. Blocks are provided to allow programmers to safely update the database with multiple commands when data integrity or consistency might be temporarily violated during the update. Programmers are encouraged to only put a block around groups of commands which might violate integrity or consistency, since use of blocks does incur some performance overhead to be discussed later.

2.3.2 Logical vs. Physical Events

In Ariel, triggering of event-based rules is based on *logical* events rather than physical events. Logical events are defined as follows. The life of an individual tuple t updated by a single transition always falls in one of the following four categories, where i, m and d represent insertion, modification (update), and deletion respectively. Superscripts $*$ and $+$ indicate a sequence of zero or more and one or more individual updates.

update type	description	net effect
im^*	insertion of t followed by zero or more modifications	insert
im^*d	insertion of t followed by zero or more modifications and then deletion.	nothing
m^+	t existed at the beginning of the transition and was modified one or more times.	modify
m^*d	t existed at the beginning of the transition, was modified zero or more times, and then deleted.	delete

The table above shows how the net effect of a sequence of updates to one tuple can be summarized as a single insert, delete or modify operation, or no operation.

We made the decision to use logical rather than physical events for the following reasons:
1. When multiple event-based rules triggered by the same event are active, execution of one rule may invalidate (e.g., delete) the data bound to another. If all binding of data to event-based rules occurs at the time the event occurs, there is no way to avoid execution of rules bound to data that is no longer valid. If events are treated as logical events as defined in the table above, rules are always bound to valid data when they execute.
2. Treating events as logical operations provides additional data integrity compared with treating them as physical operations. For example, consider the rule

define rule NoBobs
on append emp
if emp.name = "Bob"
then delete emp

The effect of this rule is to never let anyone named "Bob" be appended to the emp relation. Consider the following block of update commands:

do
    append emp(name="", age=27, sal=55000, dno = 12)
    replace emp (name="Bob") **where** emp.name = ""
end

If events are interpreted as physical operations, then this sequence of commands will not trigger rule NoBobs. However, NoBobs will be triggered if the block is treated as the following single logical event:

append emp(name="Bob", age=27, sal=55000, dno = 12)

In general, interpretation of events as logical rather than physical is expected to be more intuitive and easy to use for rule programmers, since they will only have to be concerned with *effects* of database operations, not the *expression* of them. Since many different sequences of commands can have the same effect, considering only the logical effects of updates will simplify design of event-based rules.

The above example also shows that it can be difficult to specify event-based rules to achieve a desired goal (e.g., ensuring that there is none named "Bob" in the emp relation). Hence, we recommend use of purely *pattern-based* rules whenever possible, since they will be triggered whenever any data matches a specific pattern, regardless of the event that created or modified the data. An alternative to the NoBobs rule that is purely pattern-based is the following:

define rule NoBobs2
if emp.name = "Bob"
then delete emp

This rule deletes all emp records with name "Bob" whether they are created by an **append** or a **replace** command.

until (*no rules left to run* **or** *halt executed*)
{

match
conflict resolution
act

}

Figure 1: The recognize-act cycle.

2.3.3 Rule Priorities

Each Ariel rule has a priority assigned to it which can be a floating-point number in the range -1000 to 1000. The **priority** clause is optional, and if it is not present, priority defaults to 0. Priorities are used to help the system order the execution of rules when multiple rules are eligible to run. Only rules with priority equal to the maximum of the priorities of all rules on the agenda are eligible to run.

2.3.4 The Rule Execution Cycle

Rules in Ariel are processed using a control strategy called the *recognize-act cycle*, shown in Figure 1, which is commonly used in production systems [For81].

The *match* step finds the set of rules that are eligible to run. The *conflict resolution* step selects a single rule for execution from the set of eligible rules. Finally, the *act* step executes the statements in the rule action. The cycle repeats until no rules are eligible to run, or the system executes an explicit halt.

2.3.5 Conflict Resolution Phase

The conflict resolution rule for Ariel is a variation of the LEX strategy used in OPS5 [BFKM85]. Ariel picks a rule to execute during the conflict resolution phase using the following criteria (after each of the steps, shown below, if there is only one rule still being considered, that rule is scheduled for execution, otherwise the set of rules still under consideration is passed to the next step):

- Select the rule(s) with the highest priority.
- Select the rule(s) most recently awakened.
- Select the rule(s) whose condition is the most selective (the selectivity is estimated by the query optimizer at the time the rule is compiled).
- If more than one rule remains, select one arbitrarily.

2.3.6 Act Phase

Data matching the rule condition is stored in a temporary relation called the *P-node*. In the *act* phase, the statement(s) in the **then** part of the rule are bound to the P-node for the rule by a

process of query modification [Sto75]. The modified syntax tree for the command is then passed to the query optimizer which generates an optimal query execution plan. The plan is then interpreted to carry out the command. Details of the query modification procedure will be discussed in section 7.

2.3.7 Event and Transition Conditions

One feature of Ariel that distinguishes it from most other active database rule systems is support for event and transition conditions that is fully integrated with pure pattern-based rule condition testing. Notation for specifying event-based rules was discussed previously. ARL provides a special keyword **previous** for referring to the previous value of an attribute. The value that a tuple attribute had at the beginning of a transition can be accessed using the following notation:

previous *tuple-variable.attribute*

An example of a rule with a transition condition in it is:

define rule raiseLimit
if emp.sal > 1.1 * **previous** emp.sal
then append to salaryError(emp.name, **previous** emp.sal, emp.sal)

The affect of this rule is to place the name and new/old salary pair of every employee that received a raise of greater than ten percent in a relation salaryError. Other rules could be defined to trigger on appends to salaryError to take an appropriate action, such as reversing the update, or notifying a person to verify the correctness of the update.

As an example of how pattern-based conditions and transition conditions can be combined, suppose we wished to make the raiseLimit rule specific to just the Toy department. This can be done using a normal pattern-based condition to select the Toy department, and joining the resulting tuples to the emp tuple variable in the normal fashion. A rule that does this is the following:

define rule toyRaiseLimit
if emp.sal > 1.1 * **previous** emp.sal
and emp.dno = dept.dno
and dept.name = "Toy"
then append to toySalaryError(emp.name, **previous** emp.sal, emp.sal)

Moreover, event, pattern and transition conditions can all be combined. Consider this example of a rule that uses all three types of conditions to log "demotion" of an employee in the demotions relation:

define rule findDemotions
on replace emp(jno)
if newjob.jno = emp.jno
and oldjob.jno = **previous** emp.jno
and newjob.paygrade < oldjob.paygrade
from oldjob **in** job, newjob **in** job
then append to demotions
    (name=emp.name, dno=emp.dno, oldjno=oldjob.jno, newjno=newjob.jno)

Similar to previous examples, other rules could be made to trigger when new tuples are appended to the demotions relation to take appropriate action.

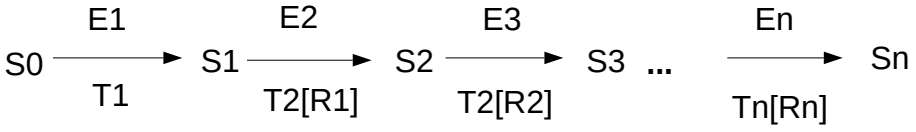
2.3.8 Transition Semantics

There are a number of different possible semantics for transition rules. We identified three possible designs which will be called level 1, 2 and 3 semantics. Level 1 semantics requires that a transition rule wake up immediately after the command that causes a transition that satisfies the rule condition. There is no need to accumulate the net effects of multiple commands to determine which transitions have occurred. Unfortunately, level 1 semantics has drawbacks including:

- It is not possible to specify a block of operations in a user transaction and ensure that no rules run inside that block, since rules must have a chance to wake up after every command.
- There can be no more than one command in a rule action, because transition rules must be run immediately after the command that triggered them. Since rules cannot run during execution of a another rule's action, this implies a single command in a rule action.
- It is unclear what to do if two or more transition rules match an updated tuple, and the first rule to execute modifies the tuple. For which new/old pair should the second rule run?

These drawbacks lead us to discard level 1 semantics.

The transition rule semantics actually implemented in Ariel is level 2 as described below. Ariel treats transitions as a set of logical events (insertions, updates and deletions). These logical events are derived by composing the physical events as they occur. Consider the following sequence of changes to the database, borrowing the notation of [WF90], where S_i is a database state, E_i is the net effect of a transition T_i, T_1 is a user-issued transition, and $T_i[R_j]$ is a transition induced by an execution of rule R_j:



The net effect of the transition from state S_l to state S_k is the composition of E_{l+1} through E_k. For example, suppose that the following emp tuple is modified by the commands and rules shown:

emp(name="Herman", age=39, sal=20000, dno=5)

User update (E1):
do
replace emp(sal = emp.sal + 1000) **where** emp.name = "Herman"
replace emp(emp = emp.sal + 2000) **where** emp.name = "Herman"
replace emp(age = 40) **where** emp.name = "Herman"
end

Rule R1 action (E2):
replace emp(sal = emp.sal + 1000) **where** emp.name = "Herman"
Rule R2 action (E3):
replace emp(sal = emp.sal - 1000) **where** emp.name = "Herman"

The above sequence of commands and triggered rules takes that data base from state S_0 to S_3 as shown in Figure 2. The net effect of this transition at states S_0 through S_3 is also indicated

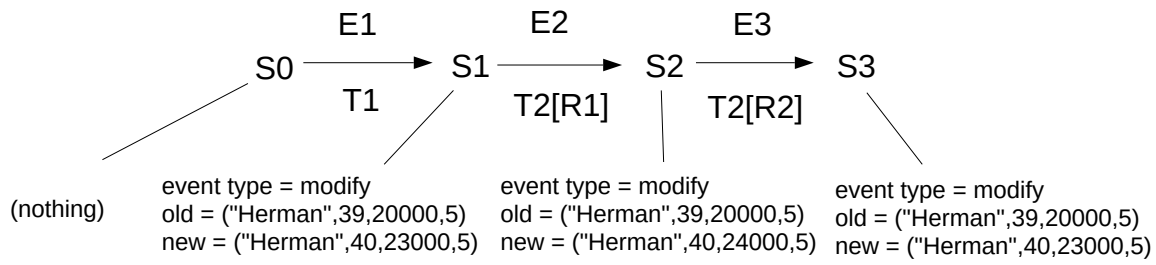


Figure 2: Transitions for example command and rules

.

in Figure 2. Rules get an opportunity to run at states S_1, S_2, and S_3. A rule with a transition condition on the emp table which is triggered for the employee "Herman" at state S_i would be bound to the token shown attached to S_i in Figure 2.

To summarize, the net effects of the changes to the database are logically updated after each user-issued command or **do ... end** block, and the changes continue to accumulate until rules terminate. After rules terminate, the changes are discarded. The old value of each old/new tuple pair accumulated is always the value that the tuple had at the beginning of the transition. This level 2 semantics allows the net effect of multiple commands on a tuple to be accumulated, and that net effect to be treated as a single logical transition. We have shown previously that the ability to combine multiple physical transitions into a single logical transition can help improve data integrity by reducing the possibility that transition constraints will be violated but still not trigger transition integrity rules.

A drawback of level 2 semantics is that an anomaly can occur, in which a transition rule wakes up, modifies the tuple bound to it, and inadvertently re-triggers itself since the net effect of the transition to the tuple still matches the rule condition. For example, consider the following rule:

define rule extraRaise
if emp.sal > 1.1 * **previous** emp.sal
then replace emp(sal = emp.sal + 500)

For values of salary greater than zero, this rule triggers itself infinitely in Ariel.

Intuitively, it would be pleasing if the condition of a transition rule referred to the following change in a tuple t:

- the change in t between the beginning of a transition and the current state,
- or, if the rule has run for t since the beginning of the transition, the change in t between the state the last time the rule executed and the current state.

This sort of semantics would help avoid problems such as the infinite self-triggering of the extraRaise rule above.

We contemplated implementing such a semantics in Ariel, called level 3 semantics, but we felt the implementation complexity would be prohibitive. It would require a log to be kept showing the value of *each updated tuple* at each database state visited during a user transition and execution of rules it triggered. In addition, there would need to be "high water marks" pointing into the log for each tuple for *each transition rule* that had run bound to that tuple. Ariel's level 2 semantics allows triggering based on the net effect of a transition, and has only moderate implementation

complexity, so we felt it was an appropriate choice. The designers of the Starburst rule system have implemented a form of level 3 semantics, albeit at substantial implementation complexity and performance overhead [WF90, WCL91]. An interesting topic for further research would be how to integrate level 3 transition rule semantics in an efficient, discrimination-network-based rule condition testing system.

2.3.9 External Functions

The ability to call external functions from within a DBMS query language is quite useful, and some form of external function interface has been implemented in several systems including ADT-INGRES [Sto86], POSTGRES [SRH90] and STARBURST [HCL+90]. In an active rule system, external functions are even more important than in a traditional database system since they allow vital communication with external processes to be performed automatically in the actions of triggered rules. Ariel supports an external function interface which allows the user to write a function in C, register the function with the DBMS using the **define function** command, and then call the function using the **execute** command. When the function is called, it is dynamically linked to the Ariel system unless it has been linked previously. The format of the **define function** command is:

define function *return-type function-name*
(*argument-list*) *file-name*

The *return-type* can be one of the built-in types of Ariel, or else **void** if there is to be no value returned.

Functions can be executed either using the **execute** command, or from withing an expression evaluation in another command. The **execute** command has the following general form:

execute *function-name* (*target-list*)
[**from** *from-list*]
[**where** *qualification*]

This command executes the function once for each tuple retrieved in the target-list. Return values are ignored in this case.

An example function to send a message to the personnel officer if someone is demoted could be defined as follows:

define function void notifyOfDemotion
(empname = c20) notifyOfDemotion.o

An example rule that makes use of this command is:

define rule notifyOfDemotion
on append to demotions
then execute notifyOfDemotion(demotions.name)

Another example function designed to be used in expressions in a command is:

define function float futureValue
(n=**float**, y=**float**, i=**float**) "futureValue.o"

This function would compute the future value of n dollars in y years at interest rate i. An example use of this function is:

retrieve (fv = futureValue(1000,10,.10))

This would simply print out the future value of $1000 after 10 years at 10 percent interest. In general, the futureValue function could be used in any expression anywhere in the target list or qualification of a command.

2.4 Rule Language Summary

ARL is a comprehensive active rule language for a relational DBMS. Important features of ARL include:

- support for production-system style programming in a DBMS, with execution semantics similar to those provided by the OPS5-LEX strategy, plus support for rule priorities and a set-oriented rule execution style,
- ability to create rules with pattern, event, and transition-based conditions,
- support for one or more data manipulation or external procedure execution commands in a rule action,
- binding of data matching the rule condition to the commands in the rule action at run-time, based on use of tuple variable names in common between the condition and action.

These features provide a powerful new capability for a relational database system, giving a foundation on which new active database applications can be built.

3 Architectural Overview

The architecture of Ariel, shown in Figure 3, is similar to that of System R [A+76] with additional components attached for rule processing. Similar to System R and other relational database systems, Ariel has a front-end consisting of a lexer, parser, semantic analyzer, and query optimizer. The back end of Ariel consists of a query plan executor, and is built on top of the storage system provided by the EXODUS database toolkit [CDF+86, RC87]. In addition to the standard front and back end components, Ariel has a *rule catalog* for maintaining the definitions of rules, a *discrimination network* for testing rule conditions, a *rule execution monitor* for managing 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. Each of these rule-system components will be discussed in detail below.

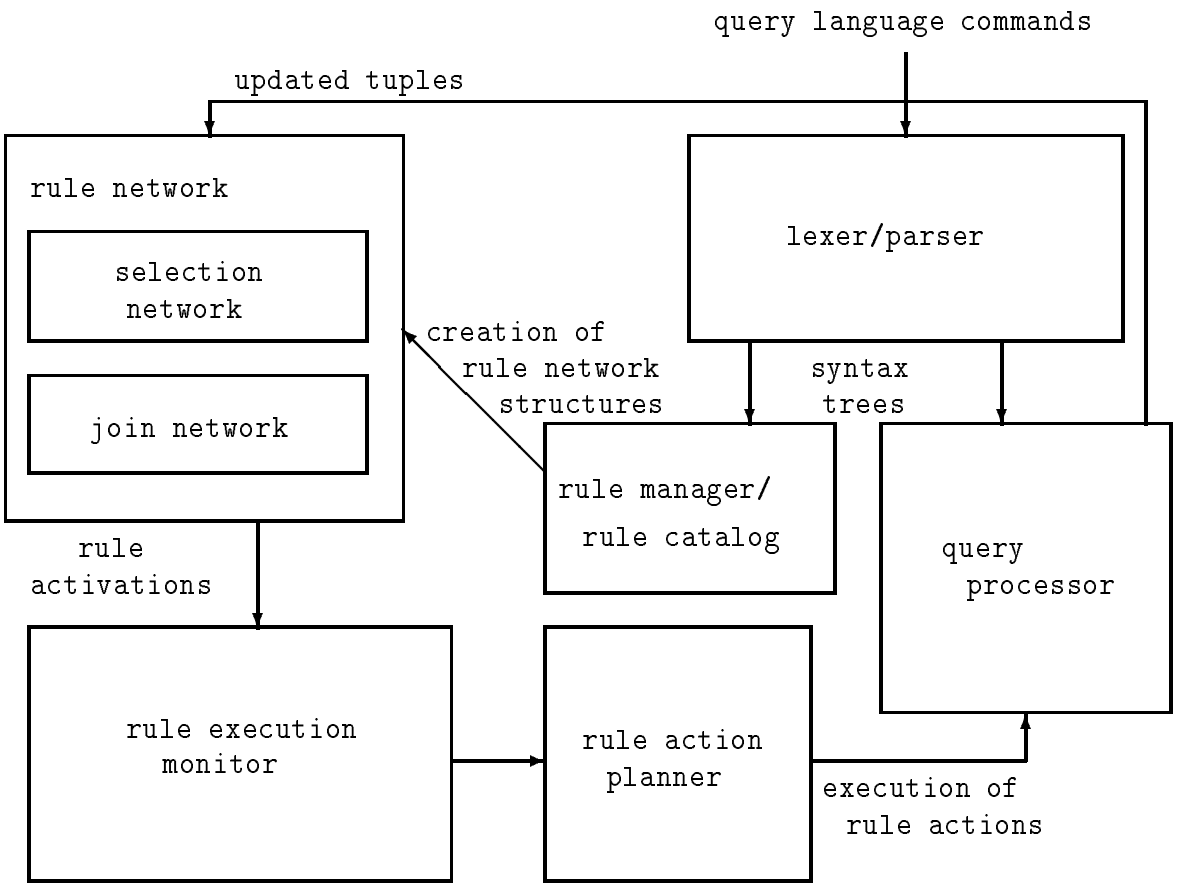


Figure 3: Diagram of the Ariel system architecture.

4 The Rule Catalog

The rule catalog is composed of a collection of Rule objects stored as persistent C++ objects (we use the persistence features of the E programming language, a persistent extension of C++ provided with EXODUS [RC87]). Each rule object contains the rule name, ruleset name, status of the rule (active or inactive), and persistent syntax tree for the rule. The persistent rule syntax tree is obtained by making a persistent copy of the syntax tree output by the parser at the time the rule is defined. The rule catalog maintains the definitions of all rules in the system, and is used whenever a rule is accessed, including the time when a rule is defined, destroyed, activated, deactivated, or triggered.

5 The Rule Execution Monitor

The rule execution monitor maintains the rule agenda, firing rules as required. The rule agenda is implemented as a priority queue, with one entry, called a *priority group*, for each group of rules

with equal priority. Within a priority group, rules are ordered such that the one whose condition was most recently matched is first.

The interface to the RuleExecutionMonitor class includes the following methods:

- **addRule** called by the rule network when a new combination of tuples matching a rule condition is found. If the rule is not already on the agenda, an activation for the rule is created, and placed at the head of the list for the appropriate priority group (a new priority group will be created if no other rule with the same priority as the added rule is active). The new combination of tuples matching the rule condition is appended to the P-node for the rule.

- **removeRule** called by the rule network when a combination of tuples that used to match the rule no longer matches. This combination of tuples is removed from the P-node for the rule. If the P-node becomes empty, then the rule instantiation is removed from the agenda.

- **runRules** called by the query executor at the end of processing a database transition. This method transfers control to the rule execution monitor, which dispatches the the most recently triggered rule from the highest priority group for execution by calling the rule action planner.

The methods described above are sufficient to allow the rule execution monitor to maintain a current list of rules eligible to run, and to assume control and run those rules at the appropriate time.

6 The 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. The main performance optimization in A-TREAT is the use of a special top-level discrimination network for testing selection conditions of rules [HCKW90]. In addition, we introduce a technique for reducing the amount of state information stored in the network, whereby α-memory nodes are replaced in some cases by *virtual* α-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. All of these techniques are discussed in more detail below.

6.1 The Top-level Discrimination Network

Efficient ways to determine which single-relation selection predicates match every new and modified tuple are important in virtually any production rule system. Selection conditions must be tested regardless of how join conditions are tested. The predicate testing problem in database rule system is defined as follows. We are given a database containing a set of n relations, R_1 ... R_n, and m production rules (triggers), r_1 ... r_m. Rules are of the form

if *condition*
then *action*

A rule condition can be an expression containing a conjunction of selection conditions and joins (projection is not allowed in rule conditions). Considering only the selection conditions of the rules, there is a collection of k single-relation predicates, P_i, $1 \leq i \leq k$. Each predicate restricts one or more attributes of a tuple t from a relation R_j where $1 \leq j \leq n$. We assume that any predicate containing a disjunction is broken up into two or more predicates that do not have disjunction, and these predicates are treated separately. The general form of a predicate purposes of this discussion is a conjunction of the following form:

$$P_i \equiv (\text{the tuple } t \text{ is in relation } R_j) \wedge C_1 \wedge C_2 \wedge ...C_q$$

where each C_j, $1 \leq j \leq q$, is one of the following:

$C_j \equiv const_1\ \rho_1\ t.\text{attribute}\ \rho_2\ const_2$
$C_j \equiv t.\text{attribute} = const_1$
$C_j \equiv \text{function}(t.\text{attribute})$

In addition, $const_1 \leq const_2$, both $const_1$ and $const_2$ are drawn from the domain of legal values for t.attribute, and ρ_1 and ρ_2 are one of $\{<, \leq\}$. Equality predicates are a special case of range predicates, but since they are so common, they are listed separately. For predicate clauses of the form "function(t.attribute)," nothing is assumed about the function except that it returns true or false.

Here are some examples of predicates on tuples of the relation emp:

emp.salary < 20000 and emp.age > 50
20000 ≤ emp.salary ≤ 30000
emp.name = "Emmett"
IsOdd(emp.dno) and emp.age = 30

In the last predicate above, IsOdd is a function that returns true if its argument is an odd number, and false otherwise.

Given the collection of predicates described above, and a tuple t, the predicate testing problem is to determine exactly those P_i's that match t. One approach to testing predicates is to use a predicate index. Many approaches to the predicate indexing problem have been developed. Below, we review the approaches proposed previously, and. We then turn to a discussion of pragmatic considerations regarding predicate indexing in a DBMS. Finally, we present the approach designed for Ariel, and give some performance measurements.

6.1.1 Review of Predicate Indexing Methods

The simplest method for testing a collection of predicates against a tuple is to store the predicates in a list, and sequentially test the tuple against every predicate in the list to find matches, with time complexity $O(n)$ where there are n predicates. A potentially more efficient method is to partition the predicates by relation using hashing, storing a list of predicates for each relation. To find the predicates matching a tuple, a hash function is computed on the relation name of the tuple to locate the list of predicates for the relation, and then the predicates on the list are tested sequentially against the tuple. If there are m relations and n predicates, and the predicates are distributed uniformly over the relations, this technique has time complexity $O(n/m)$ for finding the matches. However, in the worst case, where all predicates lie on one relation, match complexity is again $O(n)$. This technique is the one normally used in main-memory implementations of production systems.

Another predicate indexing method discussed in [SSH86, SHP88], called *physical locking*, involves treating a predicate clause like a query, and running the standard query optimizer [S+79] to

produce an access plan for the query to be indexed. If the resulting access plan requires an index scan, then special persistent markers (locks) are placed on all tuples read during the scan, and all index intervals inspected during the scan. If the resulting access plan is a sequential search, then "lock escalation" is performed, and a relation-level lock is placed on the relation being scanned. When a tuple is modified or inserted, the system collects locks that conflict with the update (i.e. all relation level locks, any locks that conflict with any indexes that were updated, and any other locks previously on the tuple). For each of the locks collected, the system tests the tuple against the predicate associated with the lock.

This algorithm has the advantage that no main-memory is needed to hold a predicate index, so theoretically, a very large number of rules can be accommodated. In addition, the algorithm makes use of the standard indexes and query processor to index predicates. However, a disadvantage to this approach is that when there are no indexes, or a large number of predicate clauses lie on attributes which do not have an index, most predicates will have a relation-level lock. This degenerate case requires sequentially testing a new or modified tuple against all the predicates for a particular relation, resulting in bad worst-case performance when the number of predicates is large. Also, the set of predicates must be stored in main memory to avoid costly disk I/O to test a tuple against a predicate when a lock for that predicate is found. This negates some of the memory-saving advantages of the algorithm. In addition, the need to set locks on index intervals and on tuples complicates the implementation of storage structures.

The final class of selection predicate indexing techniques, called *multi-dimensional indexing*, utilizes a multi-dimensional data structure for indexing region data such as an R-tree [Gut84] or R+-tree [SSH86] to index predicates. The predicates are treated as regions in a k-dimensional space (where k is the number of attributes in the relation on which the predicates are defined), and inserted into the index. Each new or modified tuple is used as a key to search the index to find all predicates that "overlap" the tuple. This technique works well when most predicates define small closed regions in the space defined by the schema of the relation from which tuples are drawn. Unfortunately, we expect that the majority of predicates in most real database rule system applications will define "slices" of this space along only one or two dimensions, not closed regions. Real relational database applications often involve relations with anywhere from one to over 100 attributes, with a large fraction of relations having from 5 to 25 attributes. Typical predicates on these relations (e.g. single-relation selection conditions in WHERE clauses of queries) normally refer to only one or two attributes, and rarely to three or four [Col89]. Spatial Data structures, particularly R-trees and R+-trees, index heavily overlapping regions like these predicates poorly, degenerating to what is essentially a sequential search of all predicates in the index.

6.1.2 Practical Considerations for Predicate Indexing in a DBMS

Numerous database rule systems have been proposed recently, including Ariel [Han89], RPL [DE88a], the POSTGRES rules system [SHP88], HiPAC [DBB+88], DIPS [SLR89], and others. We envision that applications built using systems like these will be primarily data management applications, enhanced with rules which will provide improved data integrity, monitoring capability, and some features similar to those found in expert systems.

Database rule system applications will have to handle large volumes of data (perhaps millions of records). However, we expect that the number of rules in the majority of database rule system applications will be small enough that the set of rules and data structures for rule condition testing will be small enough to fit in main memory. We believe that this assumption is reasonable because rules are a form of intentional data (schema) as opposed to extensional data (contents). Moreover,

the largest expert system applications built to date have on the order of 10,000 rules [BO89], which is few enough that data structures associated with the rules will fit in a few megabytes of main memory. More typical rule-based system applications have on the order of 50 to 1000 rules.

It is possible to concoct hypothetical applications where a tremendous number of rules are used, more than can fit in a main-memory data structure. Normally, rules in such applications have a very regular structure. This regular structure can be exploited to redesign the application so that only a few rules are used in conjunction with a much larger data table. The rules then use pattern matching to extract data from the table. For example, consider an application for stock reordering in a grocery store. The store might have 50,000 items for sale, with a relation ITEMS containing one tuple for each item. One way to implement the application would be to have one rule for each item to test whether the stock of the item is below a re-order threshold. An alternative way to implement the application would be to add a field to the ITEMS table containing the re-order threshold, and a single rule which compares the current stock level to the re-order stock level. This second implementation is clearly preferable.

It is standard practice in programming expert systems to put as much of the knowledge as possible into "facts" (e.g. frames or tuples) and as little as possible into rules. This is done because knowledge structures are more regular and easier to understand than rules. This practice will be even more important in database rule system applications, where most of the "knowledge" should be stored in the database, with minimal use of rules.

The above discussion is a partial justification for building a carefully tuned main-memory predicate index to test selection predicates of rules. We discuss such a predicate index in the next section.

6.1.3 The Ariel Selection Predicate Index

Here, we introduce a predicate indexing method tailored to the problem of testing rule selection conditions in a database rule system. The task the algorithm must perform is, given a set of single-relation selection predicates as described earlier, be able to return a list of all the predicates that match a tuple t from a relation R. We wanted the algorithm to have the following properties:

1. the ability to support general selection predicates composed of a conjunction of clauses on one or more attributes of a relation,

2. fast predicate matching performance,

3. the ability to rapidly insert and delete predicates on-line.

In the algorithm used in Ariel, the system builds an index which has at the top level a hash table, using relation names as keys, similar to high-performance implementations of production systems mentioned previously. Each entry in the table contains a pointer to a second-level index for each relation. This index maintains a list of non-indexable predicates. In addition, the second-level index contains a set of one-dimensional indexes, one for each attribute of the relation for which one or more indexable predicate clauses have been defined. All predicates clauses on an attribute which are "indexable" are entered in the index on that attribute. A diagram of the data structure implementing this strategy is shown in Figure 4.

An appropriate attribute index for use in this arrangement is one that can efficiently support *stabbing queries*, where given a point, the index can be searched to find all intervals that overlap the point. In the design of Ariel, two separate *interval indexes*, the *interval binary search tree* (IBS-tree) and the *interval skip list* (IS-list) have been developed for use as these attribute indexes.

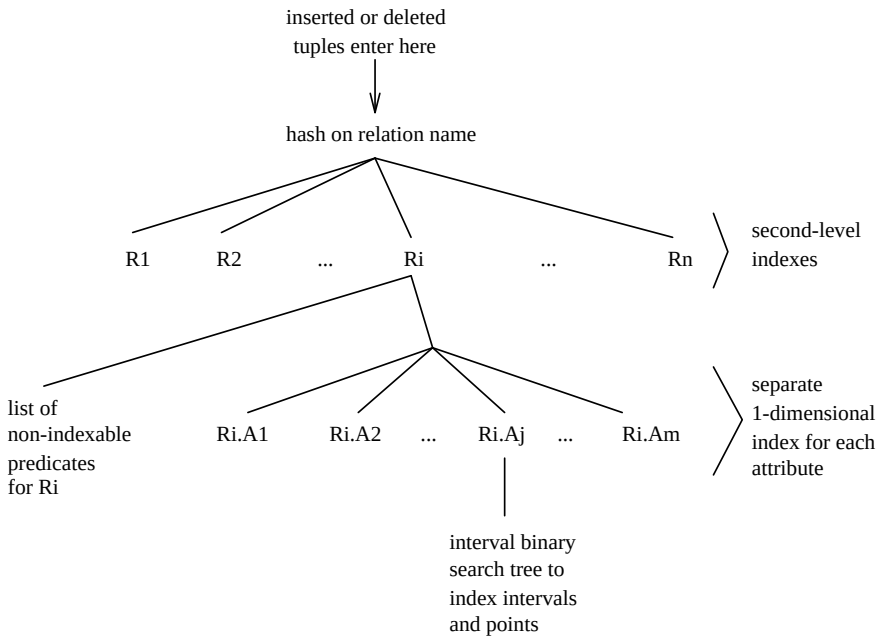


Figure 4: High-level diagram of predicate indexing scheme.

Both the IS-list and the IBS-tree support solution of stabbing queries. Given a set of n intervals, performance for both data structures is the same – $O(\log^2 n)$ time for insertion and deletion of an interval, and $O(\log n + L)$ for solution of a stabbing query, where L intervals overlap the query point.

Other interval indexes discussed in the literature, including the *segment tree* [Sam90] and the *priority search tree* were considered for Ariel, but did not meet the requirements that:

1. the index be efficiently updatable on-line,
2. a relatively straightforward implementation of the index be possible which does not require modification to index different data types, and
3. the index support fast searching to find all intervals that overlap a query point.

The segment tree does not satisfy the first requirement, and the priority search tree does not satisfy the second requirement. Both the IBS-tree and IS-list satisfy all three.

The IBS-tree and the skip-list are based on the binary search tree and the skip-list [Pug90], respectively. They involve transforming the index for point data into an interval index by augmenting the standard data structure with markers to cover each interval. Markers are placed according to an invariant such that upon searching for the location of the stabbing query point, one and only one marker will be found for each overlapping interval. An example IBS-tree and IS-list are shown in Figure 5 and Figure 6, respectively. For a complete discussion of the IBS-tree and IS-list, readers are referred to [HC90, HCKW90] and [Han91], respectively.

A. [9,19]

B. [2,7]

C. [1,3)

D. (17,20]

E. [8,12]

F. [18,18]

G. [-inf,17]

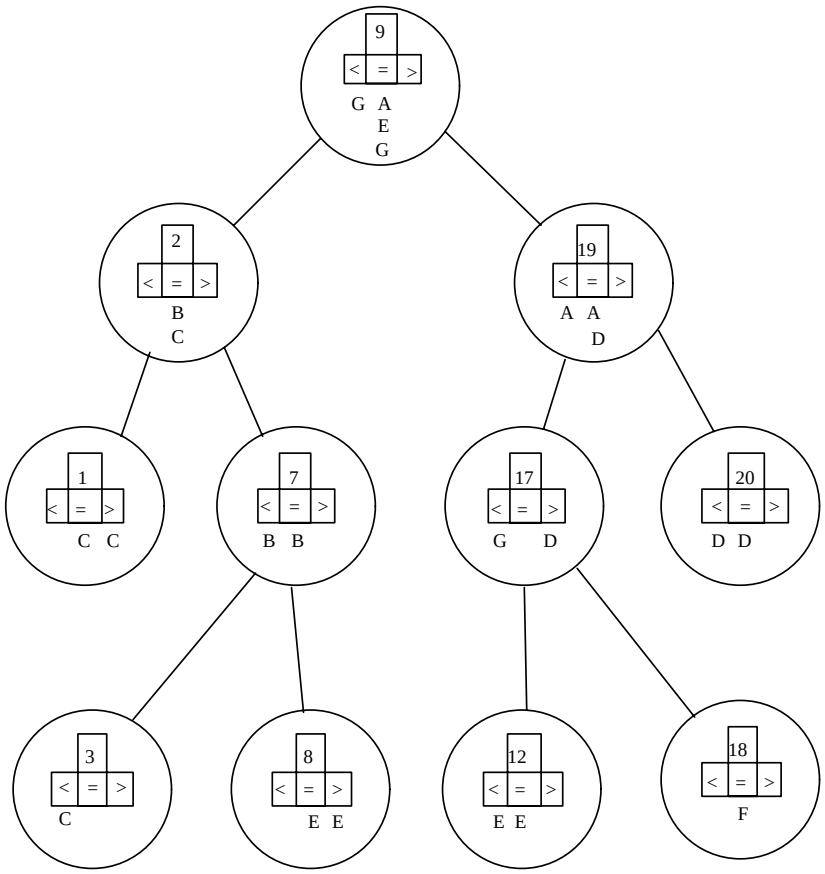


Figure 5: Example interval binary search tree for intervals shown.

6.2 Saving Storage Using Virtual α-memories

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 α-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 α-memories will be small. However, if selection conditions have low selectivity, then a large fraction of the tuples in the database will qualify, and α-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 (e.g., it is not unusual for a table to contain several gigabytes of data).

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.

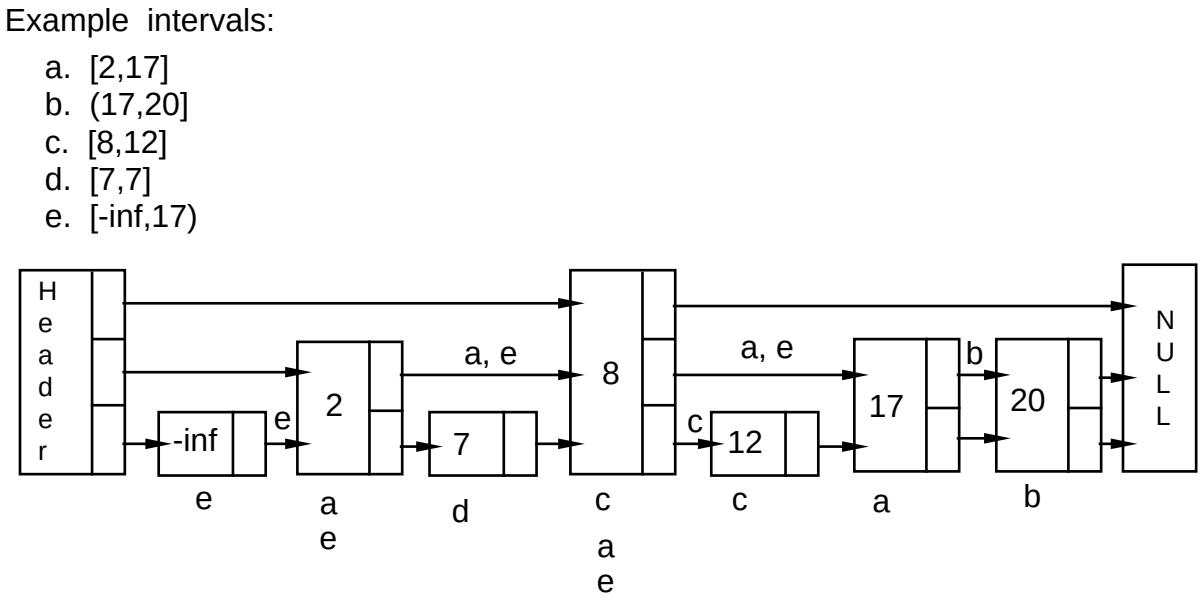


Figure 6: Example interval skip list for intervals shown.

The algorithm for processing a single insertion token t in a TREAT network containing a mixture of stored and virtual α-memory nodes is as follows. A stored α-memory node contains a collection C of the tuples matching the associated selection predicate. A virtual α-memory node contains a selection predicate 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 α-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 α-memory node A, the identifier of A is placed in ProcessedMemories and then t is joined to neighboring α-memories. When joining t to a memory node A', if A' is a normal α-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 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 α-memory node implicitly contains *exactly* the same set of tokens as a stored α-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 α-memories is being implemented in the Ariel system.

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

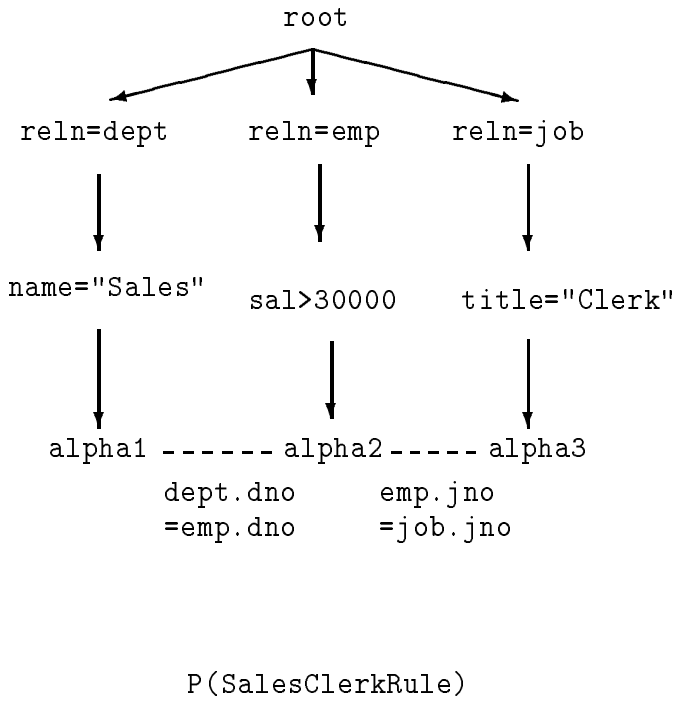


Figure 7: TREAT network for rule SalesClerkRule.

define rule SalesClerkRule
if emp.sal > 30000
and emp.dno = dept.dno
and dept.name = "Sales"
and emp.jno = job.jno
and job.title ="Clerk"
then *action*

The TREAT network for the rule SalesClerkRule is shown in Figure 7. An A-TREAT network for the rule is shown in Figure 8. The A-TREAT network is identical to the TREAT network, except that the middle α-memory node (alpha2) is virtual, as indicated by the dashed box around it. If the predicate 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.

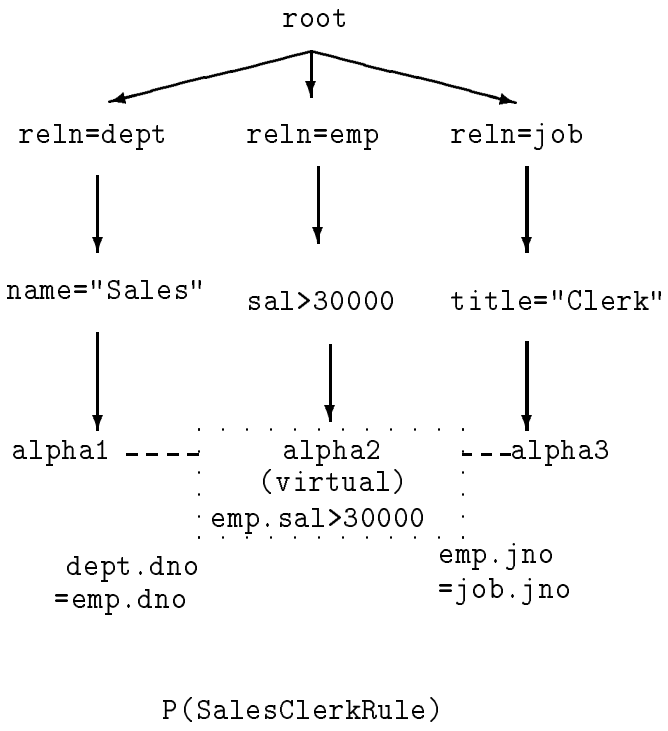


Figure 8: Example A-TREAT network.

6.3 Testing Transition, Event, and Normal Conditions Together

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.

6.3.1 Identifying Transitions

To accommodate transitions, in addition to standard + and − tokens, Ariel uses $\Delta+$ and $\Delta-$ tokens which contain a (new,old) pair for a tuple with the value it had before and after being updated. A $\Delta+$-token inserts a new transition event into the rule network, and a $\Delta-$-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 Δ-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 byte string containing 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 shown below [Ras91]:

- *Case 1:* An insertion of a tuple t followed by one or more modifications of t (im^*). The net effect of this transition is an insertion. The first insert generates an $insert^+$ token, and each modify generates an $insert^-$ followed by an $insert^+$ containing the new tuple value.

  Example:

transition	eventTokens
insert t	($insert^+$)
modify t	($insert^-$, then $insert^+$)
modify t	($insert^-$, then $insert^+$)

- *Case 2*: A tuple t is inserted, modified one or more times, and then deleted (im^*d). The net effect is nothing. Tokens are generated as in Case 1, except that the final delete operation generates an $insert^-$ token.

  Example:

transition	eventTokens
insert t	($insert^+$)
modify t	($insert^-$, then $insert^+$)
delete t	($insert^-$)

- *Case 3*: Tuple t exists prior to a transition in which it is modified one or more times (m^+). The net effect is a modification. The first modify operations generates a $modify^-$ token and then a $modify\Delta^+$. Each subsequent modify operation generates a ($modify\Delta^-$, followed by a $modify\Delta^+$.

  Example:

{t}: assertion that t exists

transition	eventTokens
modify t	($modify^-$, then $modify\Delta^+$)
modify t	($modify\Delta^-$, then $modify\Delta^+$)
modify t	($modify\Delta^-$, then $modify\Delta^+$)

- *Case 4*: Tuple t is modified zero or more times and then deleted (m^*d). The net effect is a deletion. Tokens are generated as in Case 3, except that the final delete operations generates a $modify\Delta^-$, followed by a $delete^-$.

  Example:

  {t}: assertion that t exists

transition	eventTokens
modify t	($modify^-$, then $modify\Delta^+$)
modify t	($modify\Delta^-$, then $modify\Delta^+$)
delete t	($modify\Delta^-$, then $delete^-$).

These four cases completely specify how tokens are to be created during any possible sequence of updates to a single tuple. The sequence of updates is identified at run time by using the Δ-sets [I,M], providing the information necessary to determine what type of token to create for each operation on a tuple.

6.3.2 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 α-memory nodes that are *dynamic*, i.e., they only retain their contents during the current transition.

6.3.3 Summary of Token and α-memory Types

In general, for the Ariel rule condition testing system we have identified four kinds of tokens and seven kinds of α-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,

	type of token t			
α-memory type	+	−	$\Delta+$	$\Delta-$
stored-α	insert t	delete t	insert $\pi_{new}t$	delete $\pi_{new}t$
virtual-α	insert t	delete t	insert $\pi_{new}t$	delete $\pi_{new}t$
dynamic-ON-α	insert t	delete t	insert $\pi_{new}t$	delete $\pi_{new}t$
dynamic-TRANS-α	discard t	discard t	insert t	delete t
simple-α	insert t in P-node	delete t from P-node	insert $\pi_{new}t$ in P-node	delete $\pi_{new}t$ from P-node
simple-TRANS-α	discard t	discard t	insert t in P-node	delete t from P-node
simple-ON-α	insert t in P-node	delete t from P-node	insert $\pi_{new}t$ in P-node	delete $\pi_{new}t$ from P-node

Figure 9: Table showing actions taken by each α-memory type for each token type

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 used 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 9.

In the table, "$\pi_{new}t$" represents projection of just the new part of the new/old pair contained in t. A "discard t" entry indicates that the memory node should ignore the token since the combination is not defined.

The information in this chart allows the standard TREAT algorithm to be generalized to handle normal 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 10: Example rule to illustrate query modification.

7 Optimization and Execution of Rule Actions

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.

7.1 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 a the binding between the condition and the action of the rule is made explicit through a process of *query modification* [Sto75], 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. Specificly, for a tuple variable V found in both the condition and action, every occurence 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 10. After query modification is performed on

```
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 11: Rule action after query modification.

this rule, the commands in its action look as shown in Figure 11, 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 11. Also, the **replace** and **delete** 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.

7.2 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 the uses a nested loop join, a **PnodeScan** on P, and an index scan on dept, is shown in Figure 12. 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 12.

7.3 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

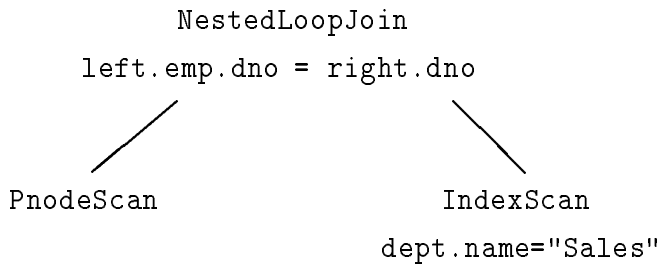


Figure 12: Example execution plan for a command in a rule action.

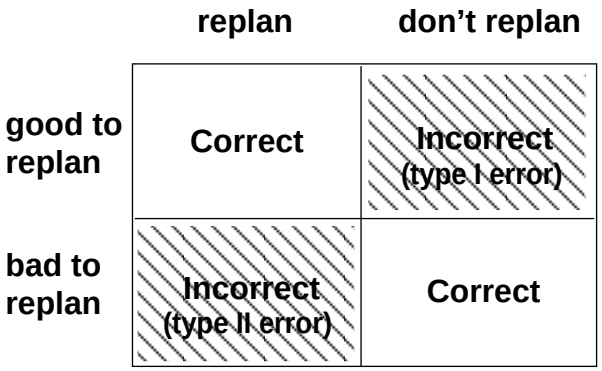


Figure 13: Outcomes of the rule action replanning decision

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. Strategies of the later type which we considered include **never reoptimize**, **heuristic**, and **cache and invalidate**. The decision whether to replan is subject to two types of errors as shown in Figure 13. If we assume that the default assumption is to replan (this is called the *null hypothesis* in statistical terminology), then not replanning when it is a good idea is a *type I* error and replanning when it is a bad idea is a *type II* error [FW80].

The different strategies are discussed here:

- **Always reoptimize.** The advantages of this approach are:
  - it *always* runs the optimal plan for execution of a command in a rule action,
  - it wastes no storage storing plans that will never be run,
  - since there are no stored plans, it is not required to build a dependency graph showing which access methods (relations and indexes) each stored plan depends on, and
  - it is straightforward to implement (only minor modifications to the optimizer are needed so it can recognize and use the **PnodeScan** operator).

The only disadvantage is that **always reoptimize** must pay the cost of running the query optimizer for every command in a rule action each time the rule is fired.

- **Never reoptimize.** This strategy compiles the plan once and never reoptimizes (unless the plan is made invalid by a change to the database schema or index structures). This strategy has low run-time overhead but may result in poor plans being run as they become out-of-date.
- **Heuristic.** This strategy decides whether to reoptimize the plan at run time using a heuristic that compares the expected cost of running the plan determined when it was first compiled, and the expected cost of running the plan given the current state of the database. The heuristic strategy may be better then never reoptimizing since it can adaptively choose whether to re-optimize. However, it suffers from an anomaly where the optimal plan could change, but the computed costs of the old plan for the old and new database are identical (e.g., if for a two-way join, one of the operands grew and the other shrank).
- **Cache and invalidate.** An alternative to the heuristic strategy is to store an optimized plan when the rule is first defined, and then have the routines that gather statistics about the data invalidate plans if the information the plans are based on gets too out of date. If a complex optimization strategy is to be implemented, this one seems most promising.

The decision of which of these strategies is best is not a simple one – the outcome can depend on numerous factors such as the time to compute an optimal query plan, the size of data tables, the type of commands in rule actions (joins vs. single-relation queries), the availability of indexes, the frequency of updates to the data, schema, and access methods, the distribution of P-node sizes when rules are triggered etc. If the data, schema, and indexes never change, then clearly a plan pre-computing strategy such as **cache and invalidate** will do better than **always recompute**. On the other hand, if a caching strategy does not recompute a plan when it should (say if the invalidation thresholds in **cache and invalidate** are set too high) then the caching strategy can run a non-optimal plan with costly results. The difference in plan execution costs could be in seconds or minutes, while the cost of reoptimization is on the order of 100 milliseconds. When a pre-planning strategy makes a (type I) error, the results can be devastating. When **always recompute** makes an error (which must be of type II), the penalty is only the time it takes to reoptimize the query. This intuition, plus the relative simplicity of **always recompute** made it the preferred choice for the Ariel implementation. However, a strategy which can drive both type I and II errors to very low levels is clearly desirable, so a detailed study of how to build such a strategy is an interesting topic for research.

8 Performance Results

Their are three main elements of Ariel's rule processing system that need to be examined from a performance standpoint:

1. the top-level discrimination network,
2. the join network, and
3. the rule action planner.

Performance results for the join network and rule action planner are not yet available. However, some performance measurements for the top-level discrimination network are presented below (see [Cha90] for a more complete performance study of the top-level network).

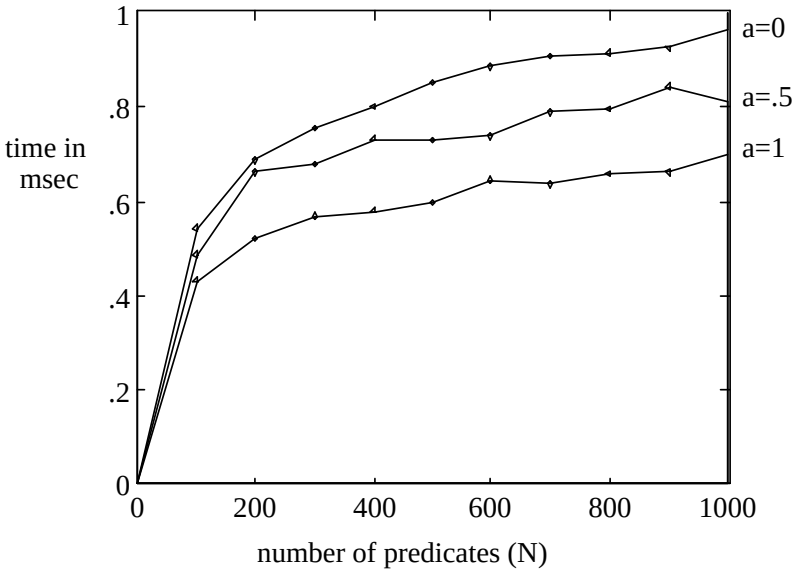


Figure 14: Average IBS-tree insertion times for a=0, .5 and 1.

To get empirical figures on the performance of IBS-trees, the algorithm was implemented in C++ on a Sun SPARCstation 1 computer. The balancing scheme using rotations was not implemented, but as with ordinary binary search trees, the tree is normally balanced if data is inserted in random order. A series of IBS trees were created which contained N predicates for N between 0 and 1,000. A fraction a of predicates were simple points of the form *attribute = constant*, and the remaining fraction $1-a$ were closed intervals. The points and interval boundaries were drawn randomly from a uniform distribution of integers between 1 and 10,000. The length of the intervals was drawn randomly from a uniform distribution of integers between 1 and 1,000. The average times to insert a predicate for values of a=0, .5 and 1, and increasing values of N are shown in Figure 14. The average insertion cost was measured as the time to insert N predicates in an initially empty index, divided by N. Since the test does not reflect any balancing cost, insertion times for balanced IBS-trees will be higher than shown in Figure 14. The average search time to find all predicates that match a value is plotted in Figure 15 for a=0, .5 and 1, and increasing values of N.

As a basis of comparison for the IBS-tree algorithm, the cost of finding the predicates that match a value by traversing a linked list of predicates and testing each one against the value is shown in Figure 16. The cost curve for sequential search is always higher than for the IBS-tree, showing that the IBS-tree has quite low overhead.

As expected, the insertion and search time curves for the IBS-tree both show logarithmic increase in search time as the number of intervals increases. The difference between the curves for the different values of a (0, .5 and 1) are small, particularly for search time.

When the IBS-tree is integrated into the overall predicate indexing scheme shown in Figure 4, predicate matching performance will depend on several factors, including:

- the fraction of predicates that are non-indexable,
- the number of attributes per relation,
- the fraction of attributes that have one or more predicate clauses,

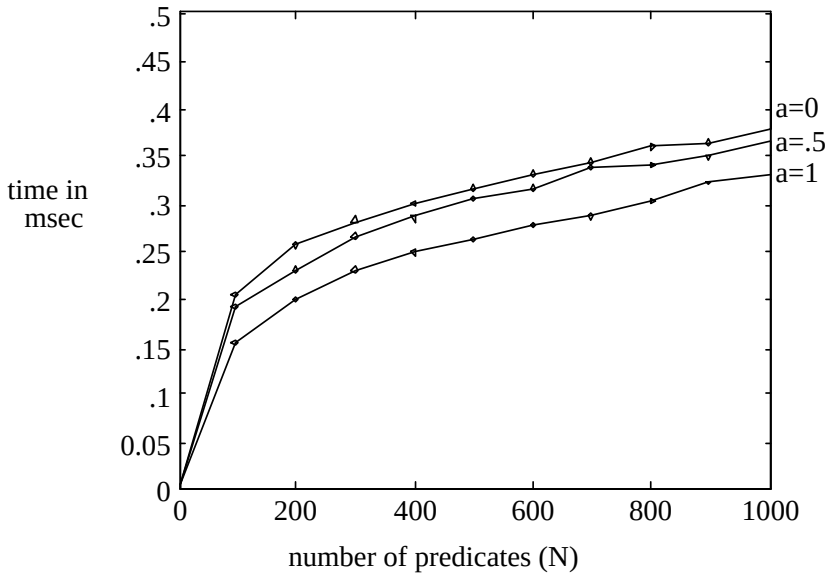


Figure 15: Average IBS-tree search times for a=0, .5 and 1.

- the number of indexable predicate clauses per attribute.

However, we can get an estimate for the time required to find matching predicates using the following assumptions:

- hash search cost = .1 msec,
- fraction of predicates that are indexable = 90%,
- cost to test a predicate against a point in sequential search = .02 msec,
- average number of attributes per relation = 15,
- fraction of attributes per relation with 1 or more predicate clauses = 1/3,
- number of predicates per relation (N) = 200 (assuming that there are 200/5 = 40 predicates per attribute, the search cost in IBS-tree for one attribute is approximately .13 msec),
- cost to test an entire predicate against a tuple when a partial match is found = .05 msec,
- number of clauses per predicate = 2,
- average selectivity of each predicate clause = .1.

The CPU usage times for operations shown above are reasonably close to the actual times for a Sun SPARCstation 1. In this scenario, the cost to search to find the partially matching predicates is the following:

cost = hash cost
+ number of attributes searched ·
IBS-tree search cost
+ non-indexable predicate test cost

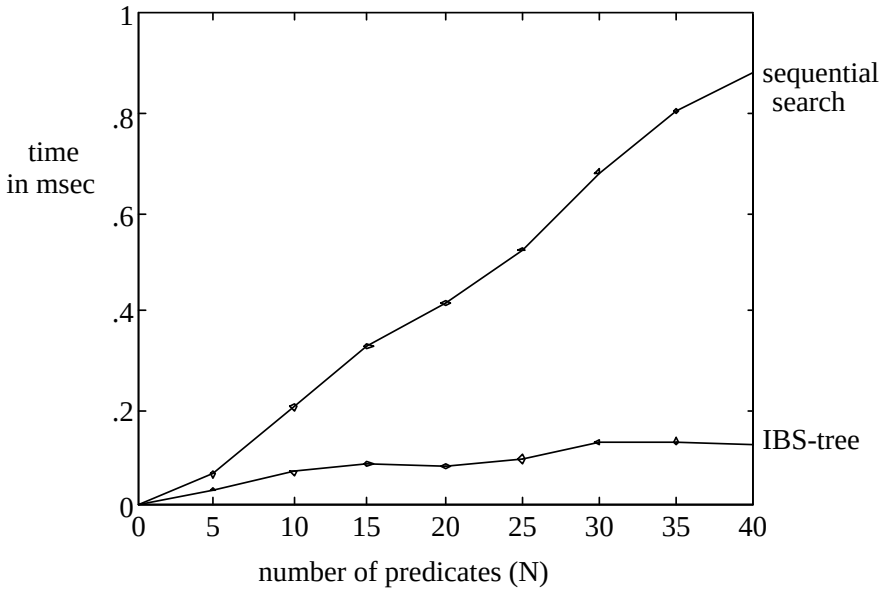


Figure 16: Predicate test cost for IBS-tree and sequential search.

This yields the following numeric expression for cost:

$$\begin{aligned} \text{cost} &= .1 + 15\tfrac{1}{3} \cdot .13 + (1 - .9) \cdot .02 \cdot 200 \\ &= .1 + 5 \cdot .13 + .4 = 1.1 \text{ msec} \end{aligned}$$

Since there are 200 predicates per relation, and the selectivity of the predicate clauses is .1, that means that $.1 \cdot 200 = 20$ predicates must be tested after the initial search. The time to test these is $.05 \cdot 20 = 1$ msec. Thus, the total time for predicate testing is $1.1 + 1 = 2.1$ msec. This is a fairly realistic number for the cost of finding all predicates that match a tuple using the algorithm presented in this paper with a moderate to large number of rules on a machine the speed of a SPARCstation 1. Given that this is a per-tuple CPU cost, the time is substantial, but should not be prohibitive. Of course, these are CPU-only costs, and any increase in CPU speed will cause the predicate testing time to scale down accordingly.

9 Implementation

Ariel is implemented using the EXODUS toolkit [CDF+86, RC87] and in particular the E programming language [RCS89], an extension of C++ with persistent objects. The current version of Ariel consists of about 28000 lines of C++/E code. We chose EXODUS and E since they were available at the time the project was started in 1988, and we wanted to focus our energy primarily on the rule processing subsystem. We felt the persistence features of E would allow us to create storage structures for relations easily, letting us avoid writing our own storage manager. Moreover, we planned to create a fairly complex persistent data structure for use as the rule discrimination network, and we felt persistent C++ would allow us to do so without writing voluminous code to read and write the data structure at system start-up and shutdown time. In addition, we wanted to take advantage of the object-oriented programming features of C++ to help us develop a complex system with hopefully less effort than would have been required in C.

```
QueryPlanOp
   Scan
      RelationScan
      SequentialScan
      IndexScan
      StoreTemporary
      PnodeScan
   Join
      NestedLoopJoin
         NestedLoopJoinIndexInner
      SortMergeJoin
   Project
```

Figure 17: Class hierarchy for query plan operators in Ariel.

In retrospect, we feel that using a persistent, object-oriented programming language was very helpful for the reasons we had hoped. The persistent objects and collections available in E made it relatively straightforward to implement the persistent discrimination network, system catalogs, and relations. The object-oriented programming features of C++ simplified and streamlined our implementations of the syntax trees output by the parser, the query plan trees, and the different types of α-memory nodes. As an example, the class hierarchy for the query plan operators in Ariel is shown in Figure 17. All three of the class hierarchies mentioned use polymorphism and inheritance extensively, simplifying them compared to a C-based implementation.

The use of E and C++ was not without difficulties. One feature of E that caused us a problem is that there is a distinction in E between regular C++ `class` types and E `dbclass` types. E thus does not have the property of *persistence orthogonality* where persistence of an object is strictly independent of its type [ABC+83]. The type of any object that is persistent in E must be declared as a dbclass and all of its sub-objects must also be db-objects. Several times we found ourselves wishing to create a persistent instance of an object (e.g., a syntax tree or a query plan) which wasn't declared as a dbclass since the need for a persistent instance of the object hadn't been anticipated. This resulted in time consuming maintenance of the software in which classes were re-defined as dbclasses, and then all sub-objects pointed to by the changed classes were changed to be dbclasses etc., with the affects rippling outward through the source code.

There are some advantages from the standpoint of language implementation to making a distinction between database types and normal types, including increased portability, ability to provide an extremely large persistent address space, and ability to easily reorganize disk-based storage. However, lack of persistence orthogonality is such a software engineering problem that we feel every effort should be made to develop persistent languages that do have persistence orthogonality [HHR91]. We are encouraged by development of at least one commercial implementation of persistent C++, Object Design's ObjectStore[Obj90], which does have persistence orthogonality, as well as research into persistent virtual-memory in the Cricket project [SZ90] which may simplify implementation of persistent programming languages.

10 Review of 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 [DE88a, DE88b], Starburst rule system [WCL91]),
- 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 [SHP88, SHP89, SRH90], HiPAC [C+89], DIPS [SLR89], Alert [SPAM91]).

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.

The POSTGRES Rule System [SHP88, SHP89, SRH90] is a sophisticated tuple-level rule system that allows triggers and integrity constraints to be defined with event and pattern-based conditions on a single tuple. It is a functioning component of the POSTGRES implementation. The POSTGRES designers have made the choice to trigger rules with single-tuple conditions as soon as the conditions of the rules are satisfied, during processing of a database update command or query. This approach makes it possible to design triggers with fine-grained, immediate response to changes, as well as implement rules which can modify tuple contents as data is being retrieved. This latter feature can be useful for implementing security and integrity features such as denying access to certain records or fields to a particular user. However, compared with rule systems with a rule agenda and scheduling mechanism that runs rules at the end of a command, group of commands, or transaction, the PRS approach is less flexible in its ability to schedule multiple rules based on recency and priority. In addition, since PRS is a tuple-level rule system, it can't take advantage of performance optimizations that can be done by set-oriented rule systems that process all data matching a rule condition together.

The HiPAC system has a sophisticated trigger model which allows specification of multiple *coupling modes* describing the time rule conditions are evaluated and rule actions are run. These include immediate, deferred, and decoupled modes for both conditions and actions [C+89]. In contrast, Ariel executes all rules in the HiPAC mode condition=immediate and action=deferred. The HiPAC design was partially implemented in a main-memory-based prototype.

RPL has a rule language based in SQL which is quite similar to the Ariel rule language. It provides an interesting model for a production-rule-like trigger language extension for SQL. However, it was implemented on top of another database system without a significant attempt to optimize rule condition testing [DE88a].

The work on the Data Intensive Production System (DIPS) describes a strategy for implementation of OPS5 on top of a relational DBMS [SLR89]. DIPS uses mechanisms based on tables of partial matches that test rule conditions differently from traditional Rete and TREAT networks. However, no clear performance measurements have been done to show which condition testing strategy is superior.

The Starburst Rule System (SRS) [WF90, WCL91] is a set-oriented rule system built on top of the Starburst extended relational DBMS. Starburst, similar to Ariel, allows specification of rules

with sophisticated transition conditions. SRS provides an elegant form of level 3 transition rule semantics as described in section 2.3.8. However, it does not use any form of discrimination network for testing rule conditions. It essentially is required to execute a query for every rule that might be affected by a particular update, which is likely to have prohibitive overhead if there are more than a handful of rules per relation.

Alert is another rule system based on top of Starburst which uses an architecture for transforming a passive DBMS into an Active DBMS [SPAM91]. Alert provides some interesting mechanisms for defining triggers using queries which return a cursor that can be accessed again to find new matching data even after an end of file (EOF) has been returned. This provides a convenient extension to relational database programming facilities to allow them to use data produced by active rules. However, Alert does not have a rule condition testing mechanism that is efficient for a very general class of rules. Their approach to testing selection conditions of rules is similar to PRS.

11 Conclusions

The Ariel project has shown that a database system can be built with an active rule system that is:

- based on the production system model,
- set-oriented,
- tightly integrated with the DBMS,
- implemented in an efficient fashion using (1) a specially designed discrimination network, and (2) 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 α-memory nodes, limiting potential optimizations.

Some commercial database rule systems already support triggers using a general predicate on a single relation (e.g., the commercial INGRES system [ING89]). A selection predicate index like the one for Ariel could be encorporated systems like this to improve performance with low risk. We hope that in the future, as experience is gained with A-TREAT style join networks, that commercial systems will be able use A-TREAT to provide the added power of triggers with joins in their conditions with fast performance.

For the future, there are a number of potential research avenues for enhancing active database systems, including:

- support for streamlined development of applications that can receive data from database triggers asynchronously (e.g., safety and integrity alert monitors, stock tickers),
- optimization of the use of storage available throughout the memory hierarchy (memory, disk, tertiary store) for storing memory nodes in a combined Rete/TREAT network augmented with virtual memory nodes,

- support for more efficient rule condition testing and execution in a DBMS using parallelism.

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.

12 Acknowledgements

I would like to thank all the people who have contributed to the Ariel project, including Yu-Wang Wang, Moez Chaabouni, Michael E. Carey, Chang-Ho Kim, Soon Chung, Y. Satyanarayana, Min Zhang, Anjali Rastogi, Indira Roy, and Hui Xu. I couldn't have done it without you! In addition, I would like to thank Abe Waksman and AFOSR for financial support of this project, and William R. Baker for giving me the freedom to conduct this research.

References

[A+76] M. M. Astrahan et al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2), June 1976.

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4), 1983. (reprinted in [ZM90]).

[BC79] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *ACM Transactions on Database Systems*, 4(3):368–382, September 1979.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming.* Addison Wesley, 1985.

[BO89] Virginia E. Barker and Dennis E. O'Connor. Expert systems for configuration at Digital: XCON and beyond. *CACM*, 32(3), March 1989.

[C+89] S. Chakravarthy et al. HiPAC: A research project in active, time-constained database management, Final Technical Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, August 1989.

[CDF+86] M. Carey, D. DeWitt, D. Frank, G. Graefe, J. Richardson, E. Shekita, and M. Muralikrishna. The architecture of the EXODUS extensible DBMS. In *Procedings of the International Workshop on Object-Oriented Database Systems*, September 1986.

[Cha89] Sharma Chakravarthy. Rule management and evaluation: An active DBMS perspective. *SIGMOD Record*, 18(3):20–28, September 1989.

[Cha90] Moez Chaabouni. A top-level discrimination network for database rule systems. Master's thesis, Dept. of Computer Science and Eng., Wright State Univ., December 1990.

[Col89] Larry Collins. Informal survey of relational database applications at Wright-Patterson AFB. (personal communication), 1989.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

[DE88a] Lois M. L. Delcambre and James N. Etheredge. The relational production language: A production language for relational databases. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 153–162, April 1988.

[DE88b] Lois M. L. Delcambre and James N. Etheredge. A self-controlling interpreter for the relational production language. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 396–403, Chicago IL, June 1988.

[Esw76] K. P. Eswaran. Specifications, implementations and interactions of a trigger subsystem in an integrated database system. Technical report, IBM Research Laboratory, San Jose, CA, 1976.

[For81] Charles L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA 15213, July 1981.

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

[FW80] J. E. Freund and R. E. Walpole. *Mathamatical Statistics*. Prentice-Hall, 1980.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, June 1984.

[Han89] Eric N. Hanson. An initial report on the design of Ariel: a DBMS with an integrated production rule system. *SIGMOD Record*, 18(3), September 1989.

[Han91] Eric N. Hanson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Proceedings of the 1991 Workshop on Algorithms and Data Structures*. Springer Verlag, August 1991.

[HC90] Eric N. Hanson and Moez Chaabouni. The IBS tree: A data structure for finding all intervals that overlap a point. Technical Report WSU-CS-90-11, Wright State University, April 1990.

[HCKW90] Eric N. Hanson, Moez Chaabouni, Chang-ho Kim, and Yu-wang Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1990.

[HCL+90] L. Haas, W. Chang, G. M. Lohman, et al. Starburst mid-flight: as the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[HHR91] Eric N. Hanson, Tina Harvey, and Mark Roth. Experiences in DBMS implementation using an object-oriented persistent programming language and a database toolkit. In *Proceedings of the 1991 ACM Conference on Object-oriented Programming Systems, Languages and Applications*, October 1991.

[ING89] INGRES Corporation. *INGRES/SQL Reference Manual*, November 1989. Version 6.3.

[MD89] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active data base management system. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, June 1989.

[Obj90] Object Design, Inc. ObjectStore technical overview, release 1.0, August 1990.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), June 1990.

[Ras91] Anjali Rastogi. Transition and event condition testing and rule execution in Ariel. Master's thesis, Dept. of Computer Science and Eng., Wright State Univ., June 1991.

[RC87] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, May 1987.

[RCS89] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. Technical report, University of Wisconsin, 1989.

[RSL89] Louiqa Raschid, Timos Sellis, and Chih-Chen Lin. Exploiting concurrency in a DBMS implementation for production systems. Technical Report UMIACS-TR-89-5, University of Maryland, January 1989.

[S+79] P. Selinger et al. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, June 1979. (reprinted in [Sto88]).

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addision Wesley, 1990.

[SHP88] Michael Stonebraker, Eric Hanson, and Spiros Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[SHP89] M. Stonebraker, M. Hearst, and S. Potaminos. A commentary on the POSTGRES rules system. *SIGMOD Record*, 18(3), September 1989.

[SLR89] Timos Sellis, Chih-Chen Lin, and Louiqa Raschid. Data intensive production systems: The DIPS approach. *SIGMOD Record*, September 1989.

[SPAM91] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. 17th International Conference on Very Large Data Bases*, Barcelona, September 1991.

[SRH90] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(7):125–142, March 1990.

[SSH86] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the First Annual Conference on Expert Database Systems*, April 1986.

[Sto75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data*, June 1975.

[Sto86] Michael Stonebraker. Inclusion of new types in relational database systems. In *Proceedings of IEEE Data Engineering Conference*, pages 262–269, 1986. (reprinted in [Sto88]).

[Sto88] Michael Stonebraker, editor. *Readings in Database Systems*. Morgan Kaufmann, 1988.

[SZ90] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. Technical report, University of Wisconsin, Fall 1990.

[WCL91] Jennifer Widom, Roberta J. Cochrane, and Bruce G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, 1991.

[WF90] Jennifer Widom and Sheldon J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990.

[ZM90] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Databases*. Morgan Kaufmann, 1990.