An Initial Report on The Design of Ariel: A DBMS With an Integrated Production Rule System *

Eric N. Hanson

Wright Research and Development Center
and Wright State University
WRDC/TXI
Wright-Patterson AFB, OH 45433

Abstract

The design and implementation strategy for Ariel, a DBMS with a built-in production rule system (a trigger system), is described. Ariel is being built with the EXODUS database tool kit. The query language of Ariel is a subset of POSTQUEL extended with a new rule language. Effort is focussed on integrating the rule system with transaction processing, and making the rule system powerful and efficient. The current implementation of the rule condition-testing mechanism in Ariel is based on a variation of the Rete network, a type of discrimination network used widely in rule-based systems tools for artificial intelligence programming. The standard Rete algorithm has been extended to improve performance in a database environment.

1 Introduction

The experience of artificial intelligence software developers over the last ten years has clearly shown the value of rule-based programming tools. Database researchers have also long recognized the potential value of rule-based systems [Esw76; BC79], and have proposed numerous "rule" or "trigger" mechanisms for database management systems (DBMSs). However, the previous work on database rule systems has not been dramatically successful, primarily because it is difficult to test database rule conditions efficiently, and to integrate execution of rules with database transaction processing. Much work remains to be done to resolve basic research issues such as the appropriate functionality for a database rule system, and efficient methods for implementing a rule execution engine in a DBMS.

The research described in this paper is an attempt to solve the difficult issues of processing triggers in a DBMS. Our approach is to build a conventional relational DBMS extended with a rule system, focusing our effort primarily on the rules component. In the spirit of things spontaneous and dynamic the system is named Ariel, after the spirit in Shakespeare's *Tempest*. The EXODUS database tool kit [CDF*86] is being used as a foundation for Ariel to simplify building the standard parts of the DBMS such as the transaction processing mechanism, access methods, and query processor. Ariel is based on the relational data model, and uses a subset of the POSTQUEL query language of the POSTGRES DBMS [SR86]. This subset of POSTQUEL has been extended with a rule sub-language different from the one used in POSTGRES [SHP88]. The rest of the paper gives some background on database rule systems, and describes Ariel's data management features, rule language, and the strategy used to implement the rule subsystem.

*This work was supported by the Air Force Office of Scientific Research under grant number AFOSR-89-0286.

2 Background

Work on rule-based systems originated in the artificial intelligence and logic programming communities. The primary examples from AI are the OPS5 system [For81] and its descendents, which use a type of discrimination network called the Rete Network [For82] to efficiently test the conditions of a collection of production rules. Another significant area of research is in logic programming, primarily the PROLOG programming language [Bra86].

Research on *database* rule systems has focused primarily on two areas: (1) deductive databases (e.g. making PROLOG work over a large database) and (2) addition of rule-based programming features to database systems. The goal of the line of research on deductive databases (1) is essentially to extend the query processor of a relational DBMS, or couple a DBMS to a PROLOG system, in such a way that recursive queries can be processed efficiently over large databases [GM78; BJ86; Ull85; ISW86]. Deductive database rule systems are passive in that they only respond to queries from users and application programs.

The goal of research into adding rule-based programming features to database systems (2) is to construct *active* database systems. Active database systems can take action in response to updates and events. Active DBMSs are qualitatively different from passive DBMSs, and hence will give rise to a new class of database applications.

The work on addition of active rule-based programming features to database systems is exemplified by the POSTGRES rule system [SHP88]. The designers of the POSTGRES rule system are attempting to build both a forward and backward chaining rule system into POSTGRES, an extensible DBMS [SR86]. The POSTGRES rule language allows rules to be defined by tagging database commands with the keyword **always**. The semantics of **always** rules are that they must appear to have just been run prior to every database transaction.

The Sybase system [How86] is the only commercial relational DBMS that currently supports a forward-chaining rules capability. However, Sybase rules can only be triggered by events such as update commands; they do not have conditions that match patterns in the data.

The HiPAC project at Computer Corporation of America [DBB*88] addresses two main problems associated with time-constrained data management: handling timing constraints in databases, and avoiding costly polling by using situation-action rules. The situation-action rules of HiPAC are a form of forward-chaining rules or triggers.

The Relational Production Language (RPL) proposed by Delcambre [DE88a; DE88b] is another forward-chaining rule language for SQL-based relational database systems. However, the RPL project has focussed primarily on language design and has not considered efficient implementation.

Work on exploiting concurrency and parallelism in database production rule systems is discussed in [SLR88; RSL89].

3 Data Management

The focus of the Ariel project is the rule system, so it was decided to keep the rest of the system as simple as possible. Other database management research projects such as POSTGRES [SR86], GENESIS [B*86], and STARBURST [S*86] are exploring ways to extend the capabilities of database systems in other dimensions, and we do not intend to compete with them. Whenever possible, we have elected to stay with time-proven technology, and use high-level tools to make the task of writing Ariel easier. Hence, we are using the relational data model and a query optimization and processing strategy similar to the one in System R [ABC*76; S*79]. The EXODUS database tool kit is being used as the foundation for Ariel, providing concurrency control, crash recovery, access methods, and query optimization services [CDF*86; GD87]. This use of EXODUS is expected to greatly simplify the process of implementing Ariel.

The query language is a simple subset of POSTQUEL that supports data definition commands, queries and updates. Language features to support POSTGRES's extensible data types, operators, access methods, and historical data are not included in Ariel's query language. Ariel will initially support only `int`, `float` and `string` data types. The `string` type will allow storage of strings of unlimited length.

4 Rule Language

The Ariel rule language (ARL) is a production rule language designed for the database environment. Ariel rules can be used to build knowledge-based systems applications, as well as to maintain database integrity constraints, and monitor database states. Rule conditions in Ariel can be based on a combination of events and pattern matching. Ariel rule conditions can also test *transition conditions*. Transition conditions have a truth value that is a function of a new database state induced by a transaction, and the previous database state. The Ariel rule system also allows definition of rules with negated condition elements. This is an important feature since analysis has shown that about 30% of the rules in rule-based expert systems contain negated condition elements [GF83].

We chose to implement production rules for Ariel rather than rules like the **always** rules of POSTGRES because we believe production rules will allow database programmers to build complex applications such as expert systems more easily than **always** rules. Production rule systems have already been proven successful for building artificial intelligence applications. In addition, implementation of production systems is already well understood. Production rule conditions that contain joins are implemented in a straightforward way using a Rete network. Testing join conditions for **always** rules using the locking scheme proposed in [SHP88] is difficult. In effect, **always** rules may be awakened unecessarily, wasting system resources.

The performance of the algorithm used to test rule conditions is very important, and we plan to investigate the performance of different condition-testing methods. Initially, we will use a version of the Rete algorithm modified to improve performance in a database environment.

4.1 Rule Syntax

Rules in Ariel have the following general form (optional clauses are surrounded by square brackets):

define rule *rule-name*
[**priority** *priority-val*]
[**on** *event*]
[**if** *condition*]
then *action*

A unique *rule-name* is required for each rule so the rule can be referred to later by the user. The **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*)]
- **retrieve** [**from**] *relation-name* [(*attribute-list*)]
- *time-specifier*

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 initially Ariel will not support aggregates in rules (other than the special negation aggregate **not**). The **then** part of the rule contains the action to be performed when the rule fires. The action can be a single database command, including any data manipulation or data definition statement. Ariel allows a *compound command* which is a **do** ... **end** block surrounding a list of other

commands. The action of a rule can contain more than one primitive command if the action is a compound command.

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 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."

The meaning of Ariel rule syntax will be discussed in more detail in the next subsection.

4.2 Rule Semantics

This section defines the meaning of Ariel rules, including a discussion of basic principles of rule execution, **on** and **if** clauses, rule priorities, transition conditions, negated conditions, and scope of tuple variable bindings.

4.2.1 Basic Principles of Rule Execution

In Ariel, commands are called *top-level* if they are not nested inside a compound command. Execution of rules is performed after every top-level command. Rules triggered by a database transaction are executed as part of that transaction.

4.2.2 ON and IF clauses

A rule may have an **on** or **if** condition, or both. **On** conditions respond to events, including time of day, and occurrence of a particular kind of database command. **If** conditions match patterns in the data.

If an **on** clause specifying a command event is present with no **if** clause, the system executes the rule's action once at the end of the top-level command in which the event occurs.

Using command event specifications, rules can be defined to trigger on **append**, **delete**, **replace**, and **retrieve** commands issued to a specific relation, and replacement or retrieval of one or more attributes of a relation as specified in *attribute-list* (a list of one or more attribute names separated by commas). An attribute is considered replaced if a value is assigned to it in the target list of a **replace** command. An attribute is considered retrieved if it appears in the target list or qualification of any **retrieve** command.

Rules can also trigger on time-based events. Ariel generates one time event every second. An **on** rule with a *time-specifier* is awakened when the specifier matches the current time. The syntax of *time-specifier* is defined as follows:

time-specifier →
| **time** = *time-list*
| **every** [INTEGER] *time-unit*
[**starting** *time-value*]
[**ending** *time-value*]

A *time-value* is of the form YY:MM:DD:HH:MM.SS using a 24 hour clock. A *time-list* is a comma-separated list of one or more *time-values*. The YY:MM:DD and SS are optional. If YY:MM:DD are not included, then the time value is matched every day on the HH:MM.SS specified. SS always defaults to 00. This syntax allows users to specify that a rule will fire at a single time, on a list of times, or on a set of times defined by a (possibly open-ended) interval and a repetition period. Ariel does not guarantee any real-time response for rules triggered by time events. The system will simply do its best to execute each rule as soon as possible after it is awakened. The action of each rule triggered by a time event is executed as a separate transaction.

A rule with only an **if** condition is triggered by an update transaction if the transaction causes any new combination of tuples to satisfy the condition. When the rule is awakened, if there is a set S of tuples that match the rule condition, the rule is executed once for S. This is in contrast to main-memory-based production rule systems which execute the rule once for each *tuple* in S. The set-oriented method was chosen since it takes advantage of the high-performance set-oriented query processor of the DBMS.

Rules with both **on** and **if** conditions are awakened when the event specified after **on** occurs, and there is data that matches the condition after the **if** clause.

4.2.3 Rule Priority

Rules can have a priority value between -1000 and 1000. The **priority** clause is optional. 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. If more than one rule is eligible to run at a time, the system chooses the highest priority rule to execute first.

4.2.4 Negated Conditions

Ariel provides a special logical aggregate function **not** for defining negated conditions. A clause of the form

not { *condition* }

can be included in a logical expression in either a rule condition or a qualification clause in a query. The **not** function is true if and only if there are no tuples that satisfy *condition*. As in aggregates in POSTGRES, binding between the condition inside the **not** and the rest of the query is accomplished by having tuple variables appearing both inside and outside the **not** refer to the same data.

4.2.5 Transition Conditions

Ariel allows definition of rules that test transition conditions using a special key word **previous**. The notation

previous *tuple-variable*

appearing in a rule refers to the previous state of the relation associated with the tuple variable (i.e. the state of the relation at the beginning of the current transaction). This way, rules can trigger on conditions which compare a new database state to the previous one. This allows definition of rules to enforce transition constraints. For any tuple variable X, there is an implicit join between X and **previous** X on tuple-id whenever **previous** X appears in the same qualification as X. The **previous** keyword is not allowed in the **where** clause of a command except if that command appears in the action of a rule.

4.2.6 Scope of Tuple Variables

Tuple variables in the condition and action of a rule are global throughout the rule. This means that tuples in the rule action already satisfy the qualifications found in the rule condition. This creates the necessary binding between the condition and action of a rule. A similar method of binding the condition and action of database rules is used in RPL for rules like the **if** rules in Ariel [DE88a]. The Sybase rule system [How86] uses a different mechanism for rules similar to Ariel's **on** rules in which special names are used to refer to sets of tuples (e.g. for an **on delete** R rule, Sybase rule actions refer to the set of deleted tuples using the special relation name **deleted**).

In Ariel, for **on** rules the following bindings hold for the different command events shown:

- **append** [**to**] R: The tuple variable R is bound to the set of tuples just appended to R.
- **delete** [**from**] R: The tuple variable R is bound to the set of tuples just deleted from R.
- **replace** [**to**] R [(*attribute-list*)]: The tuple variable R is bound to the set of new tuples just created by modifying existing tuples from R. The tuple variable **previous** R refers to the previous values of the modified tuples.
- **retrieve** [**from**] R [(*attribute-list*)]: The tuple variable R is bound to the set of tuples just retrieved from R.

For rules that specify both **on** and **if** conditions, binding of data to the condition of the rule is similar to **on** rules, except that the system also binds data to the **if** condition at the time the **on** event occurs. If any data matches the **if** condition when the **on** event occurs, the rule is packaged with the data bound to the **on** and **if** clauses, and the package is placed on the rule agenda for later execution. When the rule action is executed, the tuple variables in the **on** and **if** conditions refer to the data packaged with the rule.

The system either maintains a materialized version of the data matching the **if** condition, or runs a query just after the **on** event occurs to find the matching data. One of the two alternatives is chosen as an optimization decision. In either case, the data values bound to the rule are frozen immediately after the triggering event.

4.2.7 Semantics of Rule Actions

In general, the action of a rule can contain one or more query language commands. Any kind of command can appear in a rule action, and commands have the same effect as if executed in a user transaction, except for some differences due to binding of data between the condition and action of the rule. These differences are discussed below.

It is important to remember that since an **if** condition has the same structure as a query, more than one combination of tuples which satisfy the condition may contain the *same* tuple from a particular relation R. We call every combination of tuples that match an **if** condition a *satisfying combination*. The effects of queries and updates to data bound to tuple variables appearing in the condition of a rule are covered below for the different data manipulation commands (**retreive, replace, delete, append**).

- **retrieve:** If there is a tuple variable R appearing in the rule condition, and R also appears in a **retrieve** command in the rule action, then in that **retrieve** command, R is bound once to the R component of each satisfying combination. There may be an occurrence of the same tuple in more than one satisfying combination. In that case, R will be bound to that tuple more than once. It is the user's responsibility to remove duplicates if necessary.
- **replace:** Again, if a tuple variable R appearing in the rule condition is the target of a **replace** command in the action, R ranges over the set of satisfying combinations, and is bound once to the R component of each satisfying combination. Since the same tuple can occur in more than one satisfying combination, R may be bound to the same tuple more than once. This may cause what is known as a non-functional update, in which different values are assigned to a tuple by the same **replace** command. Non-functional updates are allowed. Only one of the assignments attempted during processing of the **replace** command will have an effect. The system makes no guarantee which one it will be. It may be that the update changes the value of the satisfying combination so that the combination no longer matches the **if** condition. Regardless of this, the combination remains in the set of satisfying combinations bound to the rule condition until the rule finishes executing.
- **delete:** If a **delete** command deletes a tuple bound to a tuple variable R, that tuple is removed from the base relation associated with R, but the satisfying combination containing that tuple remains bound to the rule condition.
- **append:** The target of an **append** is a base relation, not a tuple variable. Tuple variables are bound in an **append** command the same way they are in a **retrieve** command. An **append** does not change the contents of a rule node.

The Ariel rule language can be used to define a rich variety of rules. Some example rules defined in Ariel are given below.

```
/* Set the default desk type to metal */

define rule desk_rule1
if emp.desk = NULL
then replace emp (desk = "metal")

/*  Except set desk of managers to wood */

define rule desk_rule2
if emp.desk != "wood"
   and emp.jno = job.jno
   and job.title = "manager"
then replace emp (desk = "wood")
```

Figure 1: Example integrity rules.

```
define rule disturbance_assault
if p1.type = "assault"
   and p2.type = "disturbance"
   and abs(p2.time - p1.time) < 30
   and distance(p1.location, p2.location) < 1
   from p1, p2 in police_event
then
   append to correlations(
        event1 = p1.id,
        event2 = p2.id,
        primary_loc = p1.location,
        correlation_type=
           "disturbance_assault")
```

Figure 2: Example crime correlation rule.

```
define rule dispatch_correlations
if correlations.primary_loc = "downtown"
then execute NotifyOfCorrelation("Johnson",
                          correlations.pe1,
                          correlations.pe2)
```

Figure 3: Rule to dispatch correlations between events to appropriate detectives.

4.3 Example Rules

In the examples shown we will use the following database schema:

```
EMP(name,age,salary,desk,dno,jno)
DEPT(dno,dname,floor)
JOB(jno,title)
```

Some examples of Ariel rules to support integrity constraints are given in Figure 1.

Ariel rules can also be used to support simple knowledge-based applications. For example, consider a situation in which a database is used to help police detectives correlate events to solve crimes. The database contains a relation listing all events reported to police, as follows:

```
police_event(id, location, time,
             type, report_no)
police_case(id, crime, time, date,
            report_no, status, solved)
```

Raw input data is continually entered into the **police_event** relation. The goal of the system is to correlate different events reported to police with crimes that have been committed, hopefully identifying leads that might solve a case. For example, an assault might have been immediately preceded or followed by a disturbance caused by the assailant. An Ariel rule to try to correlate such events might try to pair up all events that occurred within 30 minutes of each other less than one mile apart. This rule could be written as shown in Figure 2. Note that a user-defined function **distance** has been used in the condition of the rule. Ariel supports user-defined functions and procedures, however for brevity we do not discuss them here.

Another rule would take potential correlations identified by the system, and dispatch them to the appropriate detectives based on location of the most important event, as shown in Figure 3. The action of the rule executes a user-defined procedure **NotifyOfCorrelation** to dispatch the correlations.

Suppose an organization had a rule that there could be no employees in a department that didn't exist. An example of an **on** rule to enforce this referential integrity constraint is shown in Figure 4. The tuple variable **dept** in the **then** part of the rule refers to the data just deleted from **dept**.

An example of a rule to enforce a transition constraint that allows only raises of 10 percent or less is shown in Figure 5. The special command **abort** used in the rule action aborts the current transaction, thus disallowing the update that raised the salary by more than 10 percent.

An example of a rule with both an **on** and **if** clause is shown in Figure 6. The **on** clause specifies that the rule should trigger on the time event **time = 14:00**. The effect of the rule is to remind employees in the Sales department to attend a staff meeting at 2:00 PM. The rule executes an external procedure **RemindAboutMeeting** to generate reminders to the employees.

Another rule that triggers on a more sophisticated time-event is shown in Figure 7. The effect of the rule is to run the procedure **CompileTradingFigures** once every hour during the workday.

5 Rule Processing

5.1 Control: the Recognize-Act Cycle

Rules in Ariel will be processed using a control strategy called the *recognize-act cycle*, shown in Figure 8, which is commonly used in production systems [For81].

The *match* step requires finding the set of rules that are eligible to run. In the *conflict resolution* step, a single rule is selected for execution from the set of eligible

```
define rule ref_integrity
on delete dept
then delete emp where emp.dno = dept.dno
```

Figure 4: Example referential integrity rule.

```
define rule raise_limit
if emp.salary > 1.1 * previous emp.salary
then abort
```

Figure 5: Example transition constraint rule.

rules. In the *act* step, the statements in the **then** part of the rule are run. The cycle repeats until no rules are eligible to run, or the system executes an explicit halt.

5.1.1 Conflict Resolution Phase

The conflict resolution rule for Ariel is a variation of the LEX strategy used in OPS5. 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.

5.1.2 Act Phase

In the *act* stage, the statement(s) in the **then** part of the rule are executed by the query processor. The query processor maintains the binding between tuple variables appearing in the rule condition and occurrences of those same tuple variables in the rule action.

5.1.3 Match Phase

The most important step from a performance perspective is the *match* phase. The initial implementation of Ariel will use a modified Rete algorithm for performing the match (see [For82] for a complete description of the Rete algorithm). In the Rete Algorithm, changes to the database cause the creation of *tokens* which are passed through the network. Tokens labeled + represent insertions and tokens labeled − represent deletions. Modifications are represented as deletes followed by inserts.

The basic components of the Rete network are the following:

- **root** node: + and − tokens are deposited at the root node, which propagates the tokens to all its successors.

```
define rule staff_mtg_reminder
on time = 14:00
if emp.dept = "Sales"
then execute RemindAboutMeeting
  ("Staff Meeting at 2", emp.name)
```

Figure 6: Example rule with both **on** and **if** clauses.

```
define rule time_rule
on every hour
   starting 08:00
   ending   17:00
then execute CompileTradingFigures()
```

Figure 7: Example rule with sophisticated time condition.

- **t-const** nodes: Tokens that arrive at one of these nodes are tested against a simple predicate. If the test succeeds, the tokens are passed onward. Otherwise, the tokens are discarded.
- **α-memory** nodes: These hold a copy of each token that has passed the test in the preceding t-const node. They also pass all tokens that arrive at them to their successor(s).
- **AND (join)** nodes: These have two inputs. Tokens arriving at one input are tested to see if they join with those at the other. If so, the joining pair of tokens is combined into one and passed onward.
- **β-memory** nodes: These hold the output of join nodes.
- **P** nodes (rule nodes): + tokens that arrive here signify that the rule should be awakened. − tokens signify that a tuple should be removed from the set of tuples currently in the active set for the rule.

A hypothetical Rete network is shown in Figure 9.

As can be seen from the figure, the Rete network is similar in structure to a collection of pre-compiled query execution plans which share common sub-expressions. The network is normally drawn with the selection conditions (t-const nodes) at the top. This is upside down compared with the way query execution plans are normally drawn (with the leaves at the bottom). The memory nodes in the network hold the value of what would be intermediate results in the evaluation of a query plan.

In Ariel's match phase, the tuples modified by a top-level command or the action of the previous rule are turned into tokens and passed through the Rete Network. If + tokens arrive at a rule node, that rule is awakened, and placed on the rule agenda. An activated rule will be executed once for the entire set of tokens at the rule node.

```
until (no rules left to run or halt executed)
{
    match
    conflict resolution
    act
}
```

Figure 8: The recognize-act cycle.

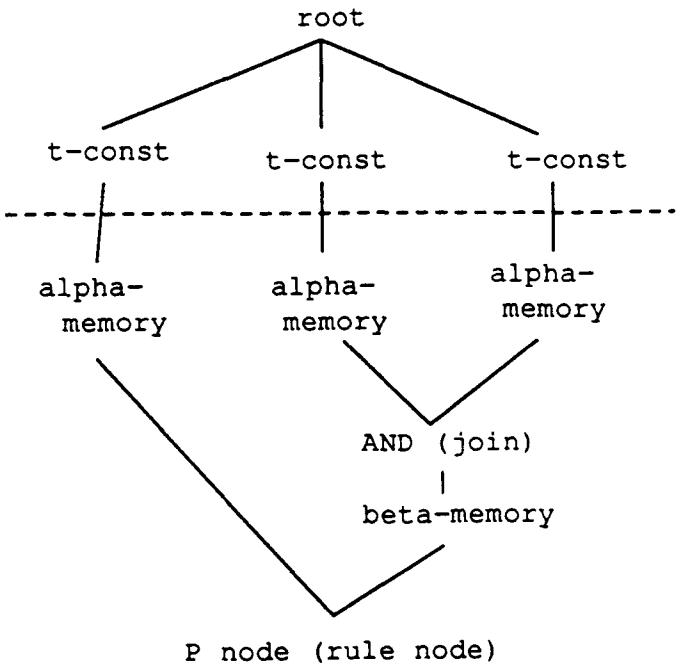

Figure 9: Hypothetical Rete network.

5.2 Alterations to Rete Algorithm to Improve DBMS Performance

In Ariel, the basic Rete algorithm will be slightly altered to improve performance in the following ways:

- The entire network, except for contents of the α-memory and β-memory nodes, is a main-memory data structure. We expect that the number of rules in the system will be small enough that the network will fit in main memory. Rules are a form of intensional data (schema), as opposed to extensional data (contents). Creation of rules is essentially the task of knowledge engineering, which is very labor-intensive. The largest rule-based systems built to date contain on the order of 10,000 rules, which is few enough that the associated Rete network will fit in memory. The memory nodes will be disk-based data, and as much of them as possible will be cached in the main-memory buffer pool. Buffer management policies for the memory nodes will be investigated. For the current implementation, memory node data will be given the same priority as other data for buffer space.
- A top-level index structure will be used after the root node to limit the number of t-const nodes that must be checked against each token as much as possible. Currently, we are planning to use a main-memory-based R-tree [Gut84] which will index on the name of the relation and other applicable features of the t-const node.

Note that no data needs to be stored for P nodes between transactions. A special case of this is that for rules that test only conditions on one relation (have no joins) there will never be any permanent state information stored in the Rete network. This gives single-relation rules a major performance advantage since disk I/O will not be necessary to test their conditions.

6 Summary and Future Work

The design of the Ariel DBMS and its integrated production rule system has been described. The goal of the Ariel project is to investigate issues in the design and implementation of active database rule systems. The design decisions that went in to the Ariel rule language were discussed, and the syntax and semantics of the language were described.

Implementation of Ariel is underway using the EXODUS database tool kit. Issues we will consider in the future include

1. building a test application in Ariel to help evaluate the design of the rule language and the performance of the rule execution engine,
2. design of an efficient top-level index for the Rete network to limit the number of predicates that must be tested against each tuple as much as possible,
3. optimization of join orderings in the Rete network structure of a database rule system, and
4. extending the rule language of Ariel with an integrity constraint sub-language to allow direct monitoring and enforcement of integrity constraints.

We feel that the first issue is the most important. It is critical to evaluate the needs of users of a database rule system to see what features are needed and what level of performance is adequate. There is currently a gap in knowledge in this area since active database rule systems are so new. A better understanding of how database rule systems will interact with conventional transaction processing requirements is needed. We encourage other researchers in the area of database rule systems to test their ideas on real applications, and to publish knowledge gained about the needs of those applications and the extent to which database rule systems satisfy those needs.

7 Acknowledgments

I wish to thank Chang-Ho Kim, Yu-Wang Wang, Moez Chaabouni, and Larry Collins for contributing to the design of Ariel, and Michael Wellman and Timos Sellis for providing valuable comments on a draft of this paper.

References

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

[B*86] D. Batory et al. *GENESIS: A Reconfigurable Database Management System.* Technical Report #TR-86-07, University of Texas, 1986.

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

[BJ86] M. Brodie and M. Jarke. On integrating logic programming and databases. In Larry Kerschberg, editor, *Expert Database Systems/Proceedings From the First International Workshop*, Benjamin/Cummings, 1986.

[Bra86] Bratko. *PROLOG Programming for Artificial Intelligence.* Addison Wesley, 1986.

[CDF*86] M. Carey, D. DeWitt, D. Frank, et al. The architecture of the EXODUS extensible DBMS. In *Procedings of the International Workshop on Object-Oriented Database Systems*, September 1986.

[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 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.

[GD87] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM-SIGMOD Conference on Management of Data*, May 1987.

[GF83] Anoop Gupta and Charles L. Forgy. *Measurements on Production Systems.* Technical Report CMU-CS-83-167, Carnegie-Mellon University, December 1983.

[GM78] H. Gallaire and J. Minker. *Logic and Databases.* Plenum Press, 1978.

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

[How86] Linda Howe. *Sybase Data Integrity For On-Line Applications.* Technical Report, Sybase, Inc., Emeryville, CA, 1986.

[ISW86] Yiannis Ioannidis, Lorna Shinkle, and Eugene Wong. Enhancing INGRES with deductive power. In Larry Kerschberg, editor, *Expert Database Systems/Proceedings From the First International Workshop*, Benjamin/Cummings, 1986.

[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.

[S*86] P. Schwartz et al. Extensibility in the Startburst database system. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, September 1986.

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

[SLR88] Timos Sellis, Chih-Chen Lin, and Louiqa Raschid. Implementing large production systems in a DBMS environment. In *Proceedings of the 1988 SIGMOD International Conference on Management of Data*, June 1988.

[SR86] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data*, 1986.

[Ull85] J. Ullman. Implementation of logical query languages for data bases. In *Proceedings of the 1985 ACM-SIGMOD International Conference on Management of Data*, 1985.