INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Distributed rule monitoring in distributed active databases

Hsu, Ing-Miin, Ph.D.
The Ohio State University, 1993

Copyright ©1993 by Hsu, Ing-Miin. All rights reserved.
DISTRIBUTED RULE MONITORING IN DISTRIBUTED ACTIVE DATABASES

DISSertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Ing-Miin Hsu, B.S., M.S.

* * * * *

The Ohio State University

1993

Dissertation Committee:
Prof. Ming T. Liu
Prof. Mukesh Singhal
Prof. Dik Lun Lee

Approved by

Prof. Ming T. Liu
Adviser
Department of Computer and Information Science
To My Parents and Family
ACKNOWLEDGMENTS

I would like to express my deepest appreciation for my advisor, Prof. Ming T. Liu, for his generosity in sharing his philosophy in research as well as in life, which was annotated with many of his personal experiences and guided me through stages of this colorful period of my time. In addition, I would like to thank Prof. Mukesh Singhal, for the inspirational discussions and his attention to details which make this dissertation possible. I would like to thank Prof. Dik Lun Lee for serving on my committee and giving timely comments at various points. Last but not least, I want to thank Dr. Yao-Nan Lien from the bottom of my heart for introducing me to the fruitful research area in active databases, which helped set the course of my research.

Most of all, I am indebted to my beloved husband and best friend, Dr. Jung Jing. Without his encouragement and support in both good times and bad times, I would not have taken on this long journey of search for excellence. Finally, no words can express my appreciation for my only entertainment source during this tight-budget period, my dearest daughters, Josephine and Joy, who provided immense satisfaction and endless love.
VITA

Dec. 2, 1958 ............. Born - Taipei, Taiwan, Republic of China

Sept. 1977 - July 1981 .. B.S., Department of Information Engineering, National Taiwan University, Taipei, Taiwan, Republic of China

Sept. 1981 - Mar. 1983 . M.S., Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

May 1983 - Nov. 1983 .. Application Programmer, Accutote, Columbus, Ohio

Nov. 1983 - Sept. 1986 . Software Engineer, Advanced Programming Resolutions, Columbus, Ohio

Sept. 1986 - March 1993 Graduate Teaching Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

PUBLICATIONS


• "Performance Study of Distributed Rule Evaluation Algorithm in Active Databases," submitted for publication. (co-authors: M. Singhal and M.T. Liu)

• "Distributed Rule Monitoring in Distributed Active Databases Revisited," submitted for publication. (co-authors: M. Singhal and M.T. Liu)

**FIELDS OF STUDY**

Major Field: Computer and Information Science

Studies in Computer Networking: Prof. Ming T. Liu
Studies in Database Systems: Prof. Dik Lun Lee
Studies in Distributed Systems: Prof. Mukesh Singhal
# Table of Contents

DEDICATION ................................................................. ii  
ACKNOWLEDGMENTS ..................................................... iii  
VITA ........................................................................ iv  
LIST OF FIGURES ........................................................ ix  

## CHAPTER  

<table>
<thead>
<tr>
<th>I</th>
<th>OVERVIEW</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Examples</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Design Issues</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Organization of the Dissertation</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>II</th>
<th>SURVEY OF LITERATURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Centralized Production Systems</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Centralized Active Databases</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Parallel Production Systems</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Parallel matching</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Multiple rule firing</td>
<td>25</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Parallel rule language</td>
<td>29</td>
</tr>
<tr>
<td>2.4</td>
<td>Parallel Processing of Recursive Rules</td>
<td>30</td>
</tr>
<tr>
<td>2.5</td>
<td>Distributed Active Databases</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>III</th>
<th>RULE DECOMPOSITION AND DISTRIBUTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Decomposition of Rule Queries</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Laws involving AND operator</td>
<td>33</td>
</tr>
</tbody>
</table>
3.1.2 Decomposition algorithm .......................................................... 35
3.2 Rule Distribution ........................................................................ 44

IV A DISTRIBUTED RULE EVALUATION ALGORITHM USING SIMUL­TANEOUS REGIONS .......................................................... 46
  4.1 Basic Ideas ................................................................................ 46
  4.2 Procedure Pseudocodes ............................................................. 47
  4.3 Sample Evolutions of the Distributed Rule Evaluation ........... 50
  4.4 Correctness Proof ..................................................................... 59

V PERFORMANCE STUDY OF THE DISTRIBUTED RULE EVALUA­TION ALGORITHM USING SIMULTANEOUS REGIONS .................. 68
  5.1 Notations and Definitions ......................................................... 69
  5.2 Derivation of Message Count .................................................. 70
  5.3 Derivation of Response Time .................................................. 72
  5.4 Numerical Results .................................................................. 76
  5.5 Simulation Data for Mean Response Time ............................... 82
  5.6 Simulation Data for Total Message Count ............................... 87
  5.7 Discussions of the Performance Data ....................................... 90
    5.7.1 Node Arrangements in a Rule Tree ................................. 90
    5.7.2 Comparison with Centralized Active Databases ............... 95

VI A DISTRIBUTED RULE EVALUATION ALGORITHM CONSIDER­ING UPDATE RATES .......................................................... 97
  6.1 Basic Idea ............................................................................... 97
  6.2 Procedure Pseudocodes ......................................................... 98
  6.3 Arguments for Correctness .................................................... 102

VII PERFORMANCE STUDY OF THE DISTRIBUTED RULE EVALUA­TION ALGORITHM CONSIDERING UPDATE RATES .................... 104
  7.1 Notations and Definitions ..................................................... 104
  7.2 Derivation of Message Count ................................................. 105
  7.3 Derivation of Parent's Update Rate ....................................... 106
  7.4 Derivation of Response Time ................................................ 106
  7.5 Discussions of the Performance Data ..................................... 107
    7.5.1 Comparison Between the Analytical Model and Simulation . 108
7.5.2 Comparison with the Distributed Rule Evaluation Algorithm
Using Simultaneous Regions ....................................................... 108

VIII CONCLUSIONS ................................................................. 115
  8.1 Summary ........................................................................... 116
  8.2 Future Work ...................................................................... 119

APPENDICES

A ALGEBRAIC LAWS FOR RELATIONAL OPERATORS ............. 121

BIBLIOGRAPHY .......................................................................... 124
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Relational Operator Tree for Rule R4 From Definition</td>
<td>37</td>
</tr>
<tr>
<td>2</td>
<td>Transformed Relational Operator Tree for Rule R4 After Step 1</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>Transformed Relational Operator Tree for Rule R4 After Step 2</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>Intermediate Relational Operator Tree for Rule R4 in Step 3</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>Transformed Relational Operator Tree for Rule R4 After Step 3</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td>Decomposed Relational Operator Tree for Rule R4</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>Message Passing in Distributed Rule Tree</td>
<td>48</td>
</tr>
<tr>
<td>8</td>
<td>Sample Distributed Rule Tree</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Legends for Evolution of the Distributed Rule Evaluation</td>
<td>53</td>
</tr>
<tr>
<td>10</td>
<td>Evolution of the Distributed Rule Evaluation</td>
<td>54</td>
</tr>
<tr>
<td>11</td>
<td>Evolution of the Distributed Rule Evaluation</td>
<td>55</td>
</tr>
<tr>
<td>12</td>
<td>Evolution of the Distributed Rule Evaluation</td>
<td>57</td>
</tr>
<tr>
<td>13</td>
<td>sibling_region_no = current_region_no</td>
<td>61</td>
</tr>
<tr>
<td>14</td>
<td>sibling_region_no &gt; current_region_no</td>
<td>61</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>sibling_region_no &lt; current_region_no</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>Sample Model of Distributed Rule</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>Computation Model for Calculating Mean Response Time Numerically</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>Analytical Mean Response Time for Various Mean Communication Delays</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>Analytical Mean Response Time for Various Mean Computation Delays</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Analytical Mean Response Time When the Mean Computation Delay is Equal to the Mean Communication Delay</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>Simulated Mean Response Time for Various Communication Delays</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>Simulated Mean Response Time for Various Computation Delays</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>Simulated Mean Response Time With Equal Computation and Communication Delays</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>Parent's Update Rate</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>Various Arrangements of m Nodes</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Message Passing in Distributed Rule Tree</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>Comparison in Parent's Update Rate between Analysis and Simulation</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>Comparison in Response Time between Analysis and Simulation</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>Comparison in Parent's Update Rate With Simultaneous Region Approach</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>Comparison in Response Time With Simultaneous Region Approach</td>
<td>113</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER I

OVERVIEW

1.1 Introduction

Recently, active databases have emerged from the integration of production systems with database management systems (DBMS's) by embedding production-style rules in database management systems [12,27,68,74]. Production systems need the capabilities of a DBMS to manage the ever increasing amount of data and rules to support more and larger applications. Database management systems, in contrast, gain a unifying mechanism for numerous database functionalities which are used to be implemented in an ad hoc manner. When production-style rules are supported, a database actively takes predefined actions (i.e., without user intervention) in response to situations that are specified on the state of the database. Traditional databases, however, function passively to react to queries or transactions explicitly invoked by users or applications.

Research interests in active databases are demonstrated by the research projects being pursued: POSTGRES [67,68,69], Ariel [27,28], Starburst [26,72,73,74] DIPS [56,57,58], IM [46], RPL [14], RDL1 [37], and Alert [53]. The above projects are all
built on top of a relational model. Active databases on top of an object-oriented model are also being pursued in HiPAC [7,12,50] and Ode [22].

Active database systems have been applied to security and authority checking [68], trigger and alerters [10,12,15,22,68], query rewrite [26], constraints maintenance [4,9,71], and materialized view maintenance [5], in addition to more applications which cannot be supported by traditional DBMS's, such as expert systems, Computer-Aided Software Engineering (CASE), and Computer-Aided Design (CAD) [15]. Since production-style rules are easy to write, potential applications for active databases also include the support of customized policies for sharing, inheritance, deduction, and version control [3].

A rule in an active database, in general, is composed of three parts, a triggering event, a rule condition, and a rule action. A triggering event can be an internal database event, such as update, delete, addition, or retrieval, or external, such as a specific time. A rule condition specifies a partial database state. The rule action consists of tasks which may be internal, such as updating the database, or external, such as launching a missile. When the triggering event occurs, the rule condition needs to be evaluated. If the rule condition is evaluated to be true, the corresponding rule action should be executed. Otherwise, no action should be taken. Evaluating every possible rule condition upon every triggering event in the system places a tremendous performance requirement on the success of the active databases. Distributing rule monitoring over multiple sites offers an opportunity to parallelize this evaluation process.
A distributed database is a database that spreads its data across multiple autonomous sites that are physically dispersed and connected via a communication network. Many such databases are now emerging for reasons of local autonomy, incremental growth, reliability, and flexibility, which fulfill the needs of enterprises situated in multiple locations with certain degrees of distributed control. The active features offered by production-styles rules are desirable for distributed databases. Moreover, rules can be used for replicated data management and semantic heterogeneity maintenance in a distributed DBMS. Along with these benefits come the problems introduced by distributed database systems, for example, global consistency, data fragmentation, and the lack of a global clock. The problems in supporting rules in distributed active databases have not received attention until recently [1,6,32].

1.2 Examples

To demonstrate some active features provided by production-styles rules in a DBMS, a sample active database and some rules that can be defined in this active database are presented in this section.

Assume a database system at a stock trading exchange is defined with the following relation schema:

STOCKS(name, class, district, open, current, high, low,...)
INDEX(name, open, current, high, low, ...)
BUYERS(tax_id, name, class, ...)
TRANSACTIONS(trans_no, name, shares, price, seller, buyer, ...)
Daily information about each stock that can be traded in this exchange is stored in the STOCKS relation, while daily information about each index, such as the Dow Jones Industrial Average of 30 Industries (DJIA) and the Standard and Poor 500 Industrial Index (S&P 500), is kept in the INDEX relation. Permanent information about each buyer allowed to trade in this stock exchange is stored in the BUYERS relation. Each trading of a stock is considered a transaction and is appended to the TRANSACTION relation after it is completed.

By following the POSTQUEL syntax proposed in the revised POSTGRES rule system, PRS II [68,69], the following are some possible rules that can be defined for this database. The ON clause introduces the triggering event, the WHERE clause the rule condition, and the DO clause the rule action.

**Example 1:** Rule R1 updates the DJIA whenever one of the 30 major stocks trades. This rule demonstrates how to propagate database updates in the form of a trigger.

\[
\text{on replace to STOCKS.current (R1) }
\]
\[
\text{where STOCKS.class = major }
\]
\[
\text{do replace INDEX (current = sum(S.current)) using S in STOCKS }
\]
\[
\text{where INDEX.name = "DJIA" and S.class = major }
\]

**Example 2:** Rule R2 is an alerter that alarms the stock exchange manager whenever the DJIA falls more than 100 points from today's opening index. A user-provided
procedure, `alarm_mgr()`, is used in the rule action to initiate an event external to the database.

```
on replace to INDEX.current (R2)
where INDEX.name = "DJIA"
and INDEX.open - INDEX.current > 100
do alarm_mgr("DJIA")
```

**Example 3:** Rule R3 maintains one kind of constraints inherent in the database semantics. It updates today's high whenever a stock price exceeds its old value. A similar rule may be defined on the INDEX relation.

```
on replace to STOCKS.current (R3)
where STOCKS.current > STOCKS.high
do replace STOCKS (high = new.current)
```

**Example 4:** Rule R4 demonstrates another type of constraint imposed by database users. R4 disallows program trading on major stocks whenever the DJIA fluctuates more than 50 points in a day.

```
on append to TRANSACTIONS (R4)
using I in INDEX
using S in STOCKS
using U in BUYERS
where I.name = "DJIA"
```
and \((I\text{-current} - I\text{-open} > 50 \text{ or } I\text{-open} - I\text{-current} > 50)\)

and \(\text{TRANSACTION.name} = \text{S.name}\)

and \(\text{S.class} = \text{major}\)

and \(((\text{TRANSACTION.buyer} = \text{U.name}\text{ or } \text{TRANSACTION.seller} = \text{U.name})\text{ and } \text{U.class} = \text{prog}))\)

do refuse append TRANSACTIONS

These four rules will be used in later chapters to demonstrate the algorithms proposed in this dissertation.

1.3 Design Issues

The design of distributed rule monitoring in active databases involves three issues:

1. Decomposing a rule condition into several parts to exploit parallelism;

2. Distributing decomposed rule conditions and base relations to sites so that rule evaluation can be carried out in parallel in multiple sites; and

3. Collecting evaluation results of decomposed rule conditions from multiple sites in a distributed environment so that the correctness of the distributed evaluation is guaranteed.

The first two issues are highly interrelated and have been addressed previously in parallel deductive databases [16,21,75] and in parallel production systems [24,30,36,47,62,64].
The third issue is identified by the authors and was published in [32] along with an integrated approach to address these three issues together.

To utilize the parallel processing power offered by a distributed system, the initial issue in distributed rule monitoring is determining how to decompose a rule into several subrules, which can be evaluated on multiple sites in parallel. The approaches proposed in the past may be considered to decompose a rule at different granularity levels, namely, at the rule set level, at the rule level, at the relation level, at the relational operator level, and at the task level.

- At the rule set level, individual rule is not decomposed. Instead, parallelism is achieved by partitioning the entire set of rules, where each partition consists of multiple rules. This approach is adopted in most of the research on multiple rule firing for parallel production systems [29,34,40,47,51,52,61,64] and also in some work on distributed active database [6].

- At the rule level, copy-and-constraint technique proposed in [62] for parallel production systems creates multiple copies of the same rule, where each copy is constrained to match a subset of the data. In the deductive database context, an equivalent idea using discriminating predicates is also independently reported in [75].

- At the relation level, rules are decomposed into single-relation subrules, where each subrule is allocated to a site and multiple sites cooperate to derive the result of a recursive rule [33].
• At the relational operator level, each test node of the Rete network [20] can be considered to be performing a relational operation, selection or join. In MAPPS [48], different test nodes of the Rete network are distributed to different processing elements.

• At the task level, Gupta [24] and Kelly and Seviora [36] proposed parallel versions of the Rete network at a fine-grain level. Each test node of the Rete network is decomposed into tasks characterized by a specific set of data, and each task is assigned to a processing element individually. Work has also been done to parallelize individual relational operators, most notably the join operator [2].

When considering rule decomposition in a distributed environment, the granularity at the task level is too fine. Since communication delays experienced in a physically dispersed system are much longer with larger variances than those experienced in parallel systems, the computation delays must be significantly longer than the communication delays in order to minimize the communication overhead introduced by the distributed architecture. The problem is totally different when the granularity is set at the rule set level. If the statistics on rules for active databases are similar to that for production systems [24], the processing times among rules are unevenly distributed. How to distribute rules at the rule set level in order to balance loads among sites has already been proved to be NP-complete [47]. How to distribute rules with consideration for both load balancing and data distribution is even tougher.
After a rule is decomposed, it is important to address the question of how to match decomposed subrules with sets of base relations in order to distribute both of them to sites. Three possible distribution philosophies are

- having a fully replicated database for the base relations [62,75],

- adapting a base relation distribution to a rule distribution [1,16,24,47,49], and

- adapting a rule distribution to a base relation distribution [31].

With a fully replicated database, the distribution algorithm has the highest freedom in choosing a site for each subrule, and the problem with the first decomposition issue is simplified. Adapting a base relation distribution to a rule distribution is popular with approaches adopted in parallel environment [1,16,29,52]. Since base relation distribution is already a very complex issue by itself and rules are much more adaptable than base relations in most distributed databases, the third philosophy which distributes rules without requiring reorganization of the current base relation distribution is a more logical choice for distributed databases.

The third issue is to determine how to collect results of subrule evaluation from multiple sites to form a global view of the entire rule condition. The criterion in this issue is correctness and consistency. A distributed evaluation algorithm is correct if and only if, first, it reports all rules whose conditions have occurred (i.e. whose conditions are true globally), and, second, it does not report rules whose conditions have not occurred. To guarantee the correctness of a global result, a consistent set of local results must be collected and combined. Traditional approaches in determining
the global state of a distributed system, such as global snapshots [8] and logical
clocks[42], are not adequate in determining a consistent rule evaluation result. Global
snapshots are suitable in detecting stable conditions, which stay unchanged once
formed, such as deadlock and process termination. Rule conditions, however, change
dynamically since database states change constantly by updates. If a rule is not
detected during the time window when the rule condition is true, it might not be
detected at all. Logical clocks, on the other hand, force an arbitrary ordering among
events that happen simultaneously on different sites. The correctness of a logical
clock approach depends on the the application and the design of the logical clocks.

The problem of supporting rules in a distributed active database has not received
attention until recently [1,6,31]. In MOBY [1], base relations are partitioned hori-
zontally in order to optimize rule evaluations. Since base relations are much less
adaptable than rules in distributed databases, this approach is more appropriate for
parallel architectures. In Widom's work [6], a rule is distributed in its entirety to a
site and accesses to data on other sites are performed through remote access. This
approach is good for processing in parallel a huge number of simple rules, each of
which only involves a few tuples of a couple of relations. For complex rules, which
involve multiple relations, it is time consuming to evaluate an entire rule whenever
one of the data referenced in the rule changes. If the statistics on rules for active
databases are similar to that for production systems [24], processing times among
rules are unevenly distributed and the matching time in response to each update to
the database is dominated by the rule with the longest processing time. Distributing
a rule in its entirety to sites does not shorten the matching time in response to each update of the database.

In this dissertation, these three issues are addressed together for complicated non-recursive rules, which not only involve multiple relations, but also are time-consuming to evaluate. Presented first is a decomposition scheme that lies between the rule level and the relational operator level. The goal of the decomposition is to find independent parts of a rule so that these parts can be distributed to different sites of the system according to the current distribution of the base relations. Assume base relations are partitioned horizontally or vertically; a distribution algorithm that will adapt rule distribution to base relation distribution is then proposed. Thereafter, a distributed rule evaluation algorithm, which guarantees the consistency and correctness of the global evaluation result by using the simultaneous region approach [60], is proposed and its performance analyzed. The performance analysis of the distributed rule evaluation algorithm is used to suggest better ways to organize a distributed rule query tree and is used to compare this algorithm with rule evaluation algorithms used in centralized active databases. The problem with this approach is the numerous messages it generates for the coordination purpose. For rules which do not become true very often, processing these messages puts an extra burden on the system without increasing productivity. A second distributed rule evaluation algorithm, which aims to reduce the number of exchanged messages, is proposed and its performance is analyzed and compared with the first approach.
1.4 Organization of the Dissertation

The remainder of the dissertation is organized as follows.

Chapter 2 is a survey of the literature in rule monitoring. Past research in centralized production systems is presented as a baseline for later discussion. Past research works in parallel production systems and parallel processing of recursive queries are then surveyed. In addition, works on centralized active databases and distributed active databases are reviewed.

Chapter 3 consists of two sections. The first section presents the rule decomposition algorithm which uses algebraic manipulations of relational operator trees. An AND operator is first introduced to identify independent parts of a rule query. The algebraic manipulations of the AND operator are then presented, followed by the rule decomposition algorithm. The second section of Chapter 3 presents the rule distribution algorithm that does not require the reorganization of the base relations.

Chapter 4 presents the distributed rule evaluation algorithm that uses the simultaneous regions approach to derive a consistent global evaluation result from local evaluation results. Examples are presented to demonstrate the operation of this algorithm. The correctness of the algorithm is proved.

Chapter 5 studies the performance of the distributed evaluation algorithm that uses the simultaneous region approach using both an analytical model and a simulation. The performance of the algorithm is studied in terms of message count and response time. Examination of the performance data reveals clues as how best to
distribute a decomposed rule query tree. The performance data are also used to compare this algorithm with the rule monitoring algorithms in centralized active databases.

Chapter 6 presents a second distributed rule evaluation algorithm, which aims to reduce the number of exchanged messages. The correctness of the algorithm is proved.

Chapter 7 studies the performance of the second distributed rule evaluation algorithm in terms of message count and response time. Both an analytical model and a simulation are used in studying the performance of this algorithm. The performance data are used to compare this algorithm with the first distributed rule evaluation algorithm.

Finally, Chapter 8 concludes with a summary of this dissertation and a discussion for future research directions.
CHAPTER II
SURVEY OF LITERATURE

This chapter surveys past research in rule monitoring in the fields of production systems, deductive databases, and active databases and in centralized, parallel and distributed environments.

Section 1 reviews the rule processing methods used in centralized production systems, namely the Rete and the TREAT algorithms, as the baseline for later discussion. Section 2 reviews the rule processing methods in three centralized active databases, POSTGRES, Ariel, and DIPS. Section 3 reviews rule processing methods proposed for parallel production systems. Section 4 reviews rule decomposition and distribution methods that are proposed for parallel and distributed deductive databases in processing recursive queries. Finally, other works in distributed active databases are reviewed in Section 5.

2.1 Centralized Production Systems

The most well-known efficient matching algorithm in production systems is the Rete algorithm [20] used in the OPS5 [19]. This algorithm, due to the fast processing and the popularity of OPS5, is often used as the baseline in the evaluation of matching
algorithms proposed for other production systems, expert database systems, and ac­
tive database systems.

Knowledge and data are arranged as follows in a production system [19]:

**Working Memory** is used to store assertions in working memory elements (WME).

Each WME consists of a type name and zero or more attribute-value pairs.

**Production Memory** is used to store inference rules in the form of productions. A
production consists of a left-hand side (LHS), which is a conjunction of condition
elements (pattern), and a right-hand side (RHS), which is a sequence of actions.

The inference engine of a production system repeatedly performs the following oper­
ations called the *recognize-act* cycle:

**Match** In this phase, the LHS's of all production rules are matched against all
WME's. Instantiations of satisfied rules, which include the rule and the WME's
that satisfy the LHS of the rule, are put into a conflict set.

**Conflict Resolution** One rule instance among all satisfied rules is selected from the
conflict set using some conflict resolution criteria.

**Rule Firing** The selected rule is fired by executing its RHS.

In mapping to relational DBMS terminology, WME's with the same type name
correspond to tuples of one base relation, whose columns correspond to attributes
specified in these WME's. Productions correspond to rules which are recently intro­
duced to relational DBMS [27,67,72]. The LHS of a production corresponds to the
condition part of the rule, while the RHS corresponds to the action part of the rule. The implicit triggering event of a production is the changes to working memory.

The Rete algorithm compiles the LHS of all defined rules into a data-driven discriminating network. At the top of the Rete network is the depository where changes to base relations come in. These changes are represented as tokens. The second level consists of single-input nodes, which test single attributes of the input token. Next is the level of two-input nodes, which test the binding of two patterns through common variables. Tokens that pass the test of single-input nodes are stored in $\alpha$-memory nodes for future feed to two-input nodes. Tokens that pass the test of two-input nodes are saved in $\beta$-memory nodes for future feed into other two-input nodes. Patterns of the LHS of one rule are linked linearly through the test nodes and the memory nodes according to the order specified in the rule. The same pattern used in multiple rules is shared by linking the same test node for all these rules. At the end is the level of terminal nodes, which hold instances of the rule whose condition has been satisfied along with the WME's that satisfy the rule, with one terminal node for each defined rule. The arrival of a token in a terminal node signifies the activation (deactivation) of the corresponding rule, which will be added to (deleted from) the conflict set.

Translating to DBMS terminology, one-input nodes correspond to the relational selection operations while two-input nodes correspond to the relational join operations. In the Rete network, all selections are done before joins. If more than one join is required in a rule, the joins are done in a fixed order as specified by the user.
The TREAT algorithm [44], originally proposed to run on a parallel production machine DADO [64], differs from the Rete algorithm in the following ways:

- There is no $\beta$-memory node. Only one-input nodes and $\alpha$-memory nodes are saved in the discriminating network.

- The join sequence for a multi-way join is constructed dynamically. Two schemes have been proposed for constructing the join sequence. The first is the lexical order, as used in the Rete algorithm. The second is the *seed order*, which starts the join from the two-input node with the newly changed WME.

### 2.2 Centralized Active Databases

POSTGRES supports rule processing with a locking method [67,68]. When a rule is defined, its condition is compiled by the standard query optimizer to produce an access plan. Any real and potential tuples read in the access plan are marked by special persistent locks, which are extensions of the original relations, with the rule ID. When a large number of tuples from a relation are marked, *lock escalation* occurs to place the lock on the entire relation, not just the specific tuples. When a tuple is modified, inserted, or deleted, all rules which have a lock on the relation and on the specific tuple are awakened. The entire awakened rule is examined. If the rule condition is satisfied, the action part is executed. Major problems with the POSTGRES approach include the complexity of the locking scheme and the high false drop rate, which is the percentage of rules awakened and examined but not fired.
In Ariel [27], a modified Rete network is proposed to process rules on top of a relational DBMS. All test nodes are maintained in main memory while all memory nodes are kept on disk. Only two types of test nodes are supported: selection (one-input node) and join (two-input nodes). The selections are done before the joins as in the Rete algorithm.

The rule indexing method proposed in [28] refines the first half of the modified Rete network in Ariel. A multi-level index is built for all rule selection predicates. The first level is a hashing on relation names. The second level consists of one-dimensional indices, one for each indexable attribute, and a list of non-indexable predicates for the relation. At the bottom level, the selection predicates on an attribute of a relation are used to construct an interval-binary search tree (IBS-tree). The IBS-tree supports not only point-type selections but also range-type selections and frequent rule modification. When a tuple is added or deleted, this multi-level index is accessed to identify selection predicates that the tuple satisfies.

In DIPS [56,57,58] a COND relation is created for each base relation to store the patterns of all rule conditions that reference the corresponding base relation. Also stored in this COND relation are tuples from this base relation that partially match these patterns and tuples with variables instantiated with values from partially matched patterns of other base relations. Compared with the Rete algorithm, a COND relation consolidates all memory nodes related to a single base relation. The relational operators considered in DIPS are selection and join.
The DIPS project is extended in [49] to address the problem of concurrent executions of multiple rule actions in support of multi-user access in a database environment. The serializability problem among concurrently executing rules is solved with a protocol based on the two phase locking (2PL) with shared Read/exclusive Write locks at relation and tuple (page) levels. All tasks associated with the execution of a rule instantiation, including retrievals from database of the tuples that satisfy the rule condition as well as changes to database from the rule action, are encapsulated in a transaction. Each retrieval or update is preceded by requesting a proper type of lock. A transaction is allowed to proceed only if the condition that satisfies the rule condition is still true in the database. Otherwise, the transaction is aborted.

2.3 Parallel Production Systems

The approaches proposed in parallel production systems have evolved from parallel matching, to multiple rule firing, to parallel rule language. Since the famous report [18] points out that the matching phase takes more than 90% of the total execution time in the early production systems, much research has been devoted to speeding up the production systems by executing the match phase in a parallel environment, while conflict resolution and rule firing phases are still executed centrally on a host. Different approaches to distributing the Rete network have been proposed to parallelize the matching phase. These approaches will be discussed in Subsection 2.3.1. The common goal of these approaches is to reduce the time spent in the matching phase in each recognize-act cycle.
Due to the improved performance of the matching phase, the need to parallelize the conflict resolution and the rule firing phases becomes apparent. The next direction in parallel production systems is to fire multiple rules concurrently. The rationale behind this direction is that by firing more rules, more WME's will be updated, which will, in turn, match more rules, and thus reduce the total number of recognize-act cycles needed to execute a production system. New problems emerge for this direction, namely the data consistency problem and the serializability problem. The data consistency problem is to identify rule instances that can be fired concurrently without interference. The serializability problem is to ensure that the result obtained from parallel rule firing is obtainable from some sequential rule firing execution. In order to deal with the new problems, the parallel matching phase is usually simplified by partitioning the entire rule set and distributing partitions among processing elements. Approaches that fall into this category are reviewed in Subsection 2.3.2.

Due to the limited speed-up gained from the parallel matching and parallel rule firing approaches, the sequentiality imposed by the production systems such as OPS5 programs is identified as the limiting factor. In the conflict resolution phase of OPS5 programs, one rule instance among all satisfied rule instances is selected using criteria such as recency and means-ends analysis. Expert system programmers have learned to use this knowledge to control the execution of the production systems they develop, making their programs highly sequential and unlikely to benefit from the parallel execution. Recent research efforts have concentrated on providing a rule programming language that is inherently parallel. Multiple rule instances are to be selected and
fired concurrently. Guaranteeing that the execution result is correct is left to expert system programmers. The approaches that belong to this category are discussed in Subsection 2.3.3.

2.3.1 Parallel matching

The DADO machine proposed in [63,64] is a massively parallel production system. A large number of processing elements (PE), on the order of hundred of thousands, are arranged in a complete binary tree, with the root node communicating with a host which provides the only interface to users. Each PE executes in one of two modes, SIMD (single-instruction multiple-data) or MIMD (multiple-instruction multiple-data) mode. Five algorithms are proposed to run parallel production systems on this machine. The first simply distributes rules among all PE’s. Each PE operates independently in MIMD mode to match WME changes broadcast to it with the rules on site. The second algorithm divides the binary tree into three levels: the upper level, the PM level, and the WM level. The rule set is partitioned and distributed among PE’s at the PM level. WME’s that are relevant to a rule are distributed in the WM-level subtrees rooted at the PM-level PE where the rule resides. The WM-level PE’s act like content addressable memories. The upper level is devoted to conflict resolution that selects one rule instance among all satisfied rule instances from PM-level PE’s. In the best case, there is one rule per PE among PM-level PE’s and one WME per PE among WM-level PE’s. If more rules and more WME’s exist than PE’s, more than one rule or one WME may be assigned to one PE. The third approach adopts the TREAT algorithm, to save partial match results between cycles,
in addition to dividing the tree into three levels. The fourth fine-grain approach maps test nodes of the Rete algorithm reversely into the binary tree. One-input selection nodes are mapped to leaf nodes, while two-input join nodes are mapped to internal nodes. The fifth approach is designed to support multiple rule firing. All satisfied rules at PM-level are fired or multiple production systems execute on a partitioned tree concurrently.

NON-VON [30] is another massively parallel systems that can support the execution of parallel Rete algorithm in multi-SIMD mode. NON-VON is organized as a tree with three types of processing elements: Large Processing Element (LPE), Small Processing Element (SPE), and Intelligent Disk Drives. The large amount of SPE's are organized as a binary tree, where each node has a small amount of RAM memory, 8-bit data path, and communication channels with neighbors. These SPE's will serve as the associative memory in the matching phase by executing the instruction broadcast to it. The small amount of LPE's have more powerful ALU and more local RAM memory than SPE's and are also organized as a binary tree. The SPE's above a certain level are connected directly with a LPE. Due to the large amount of local memory, LPE's function in multi-SIMD mode. LPE's are connected with each other through a LPE network which is connected with a host. Intelligent Disk Drives are connected to the LPE's. In NON-VON, the entire rule set is partitioned and distributed among the LPE's at a fixed level. Each intra-relation pattern of the LHS of a rule is distributed to a descendent SPE of the LPE on which the rule resides. The large amount of single-input nodes in the Rete algorithm is examined in parallel by
the associative memory offered by the SPE's. Two-input nodes in the Rete algorithm are processed in parallel in multiple LPE’s which execute in multi-SIMD mode.

A. Gupta pointed out that DADO is not very effective in executing OPS5 applications in parallel [23,24] because rule actions in OPS5 do not have global effects and only update a small number of WME's that affect a small number of production rules. Since a small number of productions are affected, only a few of the large number of PE’s perform useful work in each cycle. Furthermore, he pointed out in [24] that the performance bottleneck in parallel matching is two-input nodes with long processing time and large variance due to fluctuating sizes of the input memory nodes. The same reasoning applies to NON-VON [24].

In contrast to the massively parallel approach adopted in DADO, A. Gupta proposed in [24] a multiprocessor architecture with much fewer but high-performance PE’s to parallelize the Rete network at a fine granularity. The multiprocessor proposed consists of 32 to 64 high-performance PE’s which are connected to shared memory through shared busses. Each one-input node and each instantiation of a two-input node are assigned to a PE by a centralized software or hardware task scheduler. A product following this proposal is the ParaOPS5 [25], which is an optimized C-based parallel implementation of OPS5 for shared-memory multiprocessor. Each task in ParaOPS5 executes about 100 instructions.

Oflazer [47] agrees with Gupta that massively parallel systems such as DADO are not good approaches for parallel production systems due to the locality presented in production rules. Instead, he proposed that the entire rule set be partitioned
among a small set of high-performance processors [47] so that the load among all PE's is balanced. He formulated the partition problem as follows. An execution run of a production system is $T = \langle e_1, e_2, \ldots, e_t \rangle$, where $t$ is the number of changes performed during the run. Each $e_i$ is a set of productions which are processed in cycle $i$. Associated with each $e_i$ is a cost function, $c_i$, that captures the time cost associated with processing every production rule in $e_i$. Given a partition of $n$ production rules among $K$ processors $\Pi = (\Pi_1, \Pi_2, \ldots, \Pi_K)$, the cost of executing the production system with this partition is

$$Cost_{T,\Pi} = \sum_{i=1}^{t} \max_{1 \leq j \leq K} \left( \sum_{p \in e_i \cap \Pi_j} c_i(p) \right)$$

Then the partition problem becomes a minimization problem that finds a partition that minimizes $Cost_{T,\Pi}$. Since the minimization problem is proved by Oflazer to be NP-complete, a simulated annealing algorithm is proposed to approximate the best solution. A parallel matching algorithm, which stores all partial match results among all combinations of condition elements in order to minimize variation among processing different rules, is proposed to support the partitioned rule set.

Since it has been recognized that different rules require different amounts of time to process and the main performance bottleneck in the Rete network is the two-input nodes that match a large amount of WME's, the DRete algorithm proposed in [36] attacks the two-input node problem. DRete is a distributed Rete algorithm run on a multiprocessor CUPID. The match phase is partitioned at the token-to-token comparison level. Two-input nodes and memory nodes are replicated to provide parallelism. Memory nodes are replicated so that each copy is feeding input to one
two-input node. Two-input nodes are replicated so that each copy contains one token, from either the left or the right input memory node. An algorithm is devised to process newly arrived token on any two-input node copy. Multiple two-input node copies, therefore, can be matched in parallel in multiple PE's. A load balancing algorithm is also proposed so that copies of the same two-input nodes are not likely to be assigned to the same PE.

The copy-and-constraint technique [62] addresses the uneven processing time among rules differently. Hot spot rules which match many WME's and, therefore, require more processing time, are identified. For hot spot rules, multiple copies of the same rule are created, where each copy is constrained to match a subset of the WME's that are matched by the original rule. However, care must be taken to ensure that the union of the matched results from all copies is equal to the matched result of the original rule processed at a single site.

2.3.2 Multiple rule firing

In the cornerstone paper of multiple rule firing approaches by Ishida and Stolfo [35], a data dependency graph is introduced to analyze the interference between rules in order to fire multiple rules in a parallel environment with a Control Unit (CU) and multiple Processing Elements (PE's). Rules are interfering when the result of firing these rules concurrently is not equal to the result of firing the same set of rules in some sequential order. To fire multiple rules concurrently, the entire set of rules is partitioned and distributed among CU and PE's, while the WM is stored in the
shared-memory for the shared-memory multiprocessor and is duplicated in every node for the shared-nothing multiprocessor.

In [39,40,41,45], Kuo et al. use the data dependency graph from [34,35] to determine the compatibility between any two rules. All compatible rules are executed concurrently in the rule firing phase. However, executing only compatible rules does not guarantee a correct result. The concept of context, which has its root in stepwise refinement methodology, is introduced to guarantee the correctness of the result. Each context is classified as converging, parallel non-converging, or sequential. Different methods are used in handling different type of contexts to achieve parallelism.

For the match phase, rules are partitioned and distributed to processing nodes, each with its own Rete network compiled from the rules assigned to the site. Two partition schemes studied are partition-by-context and round-robin. The partition-by-context scheme maps all rules for one context to the same processing element. The round-robin scheme assigns rules to nodes in round-robin fashion. The performance study in [39,40,41,45] shows that the partition-by-context scheme achieves parallelism while performance using the round-robin scheme deteriorates as the number of nodes increases due to the increased number of messages exchanged.

In [61], Srivastava et al. propose static and dynamic approaches to support multiple rule firing in a multiprocessor environment. The static approach suggests partitioning rules into non-interfering sets and distributing sets to different processing nodes. The dynamic approach suggests a locking scheme that allows multiple rules to
be matched and/or executed concurrently at multiple sites by issuing different types of locks on the accessed data.

In [51], J. Schmolze formulates a formal solution to address the serializability problem for multiple rule firing in synchronous parallel production systems. The solution builds upon the proposal by Ishida and Stolfo [35]. For the rule decomposition part, rules are grouped into contexts. At any given time, only rules in one context can be active and qualify to be fired when its LHS is satisfied.

The PARS system [52] proposes an asynchronous multiple rule firing model in a no-shared memory systems, where processors communicate with each other by passing messages over an interconnection network. Each rule is assigned to exactly one processor. All WME's of the classes that might match the rule's LHS are allocated to the same processor. Some base data might be duplicated in multiple processors. The recognize-act cycle will be executed at every processor asynchronously. Due to the serializability problem introduced by firing multiple rules, synchronization messages are sent in the rule firing phase and two new phases are added to the recognize-act cycle. A disable phase is added between the conflict resolution and the rule firing phases to send a disable message to processors which might have interfering rules. Another enable phase is added after the rule firing phase to enable the rules disabled in the disable phase. A double acknowledgment protocol is used in the rule firing phase to guarantee consistency of the distributed WME's and, hence, the serializability of the multiple rule actions that may be executed in parallel.
In [29], Harvey et al. propose that an application is explicitly decomposed hierarchically into multiple levels of independent jobs which can be executed in parallel, instead of just one-level contexts as suggested in [40,51]. The decomposition can stop at any proper granularity, not limited to just a whole rule or a context. Also, the WME's, as opposed to the rules, are partitioned and distributed to multiple sites to achieve parallelism.

The MAPPS proposed in [48] is a multiprocessor system designed for parallel production systems that exploits parallelism at three levels: concept level parallelism, inter-phase parallelism, and intra-phase parallelism. The processing elements are arranged into a three stage pipeline, T-module, A-module, and C-module. The PE's at the T-module are responsible for executing one-input test nodes of the Rete algorithm. The PE's at the A-module are arranged into clusters and execute two-input test nodes of the Rete algorithm in parallel while the memory nodes of the Rete algorithm are stored in the cluster memory shared by PE's of the same cluster. It is the programmer's responsibility to partition the rule set into mutually exclusive subsets and allocate each partition to a cluster. The PE's in the C-module are arranged into a binary tree and the rule instantiations emerging from the A-module are deposited into the leaf nodes of this binary tree. A prominent rule from each cluster is selected using the usual conflict resolution criteria. Then interference among rules is analyzed at run time to remove incompatible rules at each level of the binary tree. All rules reaching the root node of the C-module are fired concurrently by executing their
actions and sending the WME updates through an inter-module bus to the PE's in
the T-module.

2.3.3 Parallel rule language

The CREL system proposed in [38] offers a comprehensive approach to the parallel
rule execution. First, a parallel rule language is defined with identical syntax with
OPS5 except that the recency criterion in the conflict resolution phase is removed and
the selection phase is made undeterministic. Conceptually, CREL fires a single rule
instantiation per cycle but several compile-time and run-time techniques are taken to
speed up the processing. Among the compile-time techniques, static data dependency
analysis, context variables, and copy-and-constraint are used to partition a rule set
into clusters so that rules in different clusters can be fired concurrently. Run-time
techniques include join-level match parallelism by performing concurrent multiway
joins, one for each α-memory update.

Besides removing the recency criteria, the PARULEL language in [65] proposes to
use meta-rule to redact unwanted rule instances from the conflict set. All remaining
rule instances in the conflict set are fired concurrently. They reason that the ap­
proaches taken in other multiple rule firing are based on syntax and are pessimistic,
and, therefore, the speed-up obtained is limited. By providing inherent parallel rule
semantics, expert system programmers are given the opportunity to fully exploit the
parallelism in the applications, and if necessary, the expert system programmer can
still program sequentiality in through meta-rules.
2.4 Parallel Processing of Recursive Rules

In [16], Dong does not decompose the recursive rule at all. The same rule is run on every site while each site has a different subset of the base relations. The base relations are decomposed and distributed according to the natural partition among the data or the required size of the base relations in order to optimize rule execution.

The discriminating predicates technique reported in [11,75] is equivalent to the copy-and-constraint technique as proposed in production systems [62]. Here, the same rule is distributed to multiple sites and the subrule at each site is appended with a discriminating predicate. Through the introduction of the discriminating predicates, the scope of data accessed by each subrule is limited and the load among processors can be balanced. However, the discriminating predicates are at the discretion of the rule designer and can not be generated automatically. A similar idea is also discussed in [21].

In [33], a recursive rule is decomposed into single-relation subrules, each of which is governed by an Evaluate process on a different processor. Processes communicate by message passing to exchange intermediate results.

2.5 Distributed Active Databases

Moby [1] is a proposed architecture for distributed expert database system. To improve performance and balance load, base relations are partitioned horizontally based on existing rules. The partitions are distributed to processing nodes, while the entire Rete network is duplicated at every processing node.
In [6], an orthogonal mechanism is used to integrate the entire recognize-act cycle into transaction processing in distributed and parallel database environment. The parallelism is achieved by performing multiple recognize-act cycles, not just the match phase, concurrently on multiple sites. The recognize-act cycles and each phase within a cycle proceed asynchronously. The correctness of the result is guaranteed by ensuring the integrity of the data through a locking scheme. Different types of locks are issued by each phase on each data item accessed to coordinate between multiple rule condition evaluations, between multiple rule action executions, and between rule condition evaluation and rule action execution, which are being performed concurrently at different sites. Specifically, the parallelism in the match phase is achieved by distributing base relations among sites. A base relation is neither fragmented nor replicated. A rule in its entirety is assigned to the site where its triggering base relation resides.
CHAPTER III

RULE DECOMPOSITION AND DISTRIBUTION

3.1 Decomposition of Rule Queries

The goal of evaluating a rule condition is to determine whether the rule condition has occurred, which can be represented as a true/false value, and, when it occurs, to pass to the rule action a subset of the derived data, which is called the binding [12] between a rule condition and its corresponding rule action. Since rule conditions introduced in relational DBMS's have taken the form of database queries, a rule condition restricted to its binding with the rule action can be easily converted to a relational expression. Optimization of a relational expression by algebraic manipulations of its corresponding relational operator tree has been studied in detail [70]. The principle idea is to move selections and projections as far down the tree as possible. Our decomposition algorithm will follow the same principle. Moreover, parts of a rule condition might simply be existential qualification checks, which neither use data generated from the triggering event nor generate any data for the rule action. The existence of these independent parts of a rule condition offers an excellent opportunity to process a rule in a distributed environment with minimum communication overhead.
Let a rule query be the relational expression derived from a rule condition projected to extract data required by the binding with the rule action. The outcome of evaluating a regular database query is a relation that satisfies the relational expression. The outcome of evaluating a rule query, however, is a relation satisfying the relational expression and a logical value. A relational expression is considered logically true if the derived relation is not empty; otherwise, it is considered false.

A new AND operator is introduced to capture this difference between rule queries and regular database queries. The AND operator is a binary operator with relational expressions as operands. The relational value of an AND expression is a relation that consists of two disjoint relations, each corresponding to an operand relation. The logical value of an AND expression is true when both of the operand expressions are true. The symbol $\land$ will be used to denote the AND operator in the rest of this paper, while conventional notations are used for existing relational operators, such as selection($\sigma$), projection($\pi$), Cartesian product($\times$), join($\Join$), union($\cup$), intersection($\cap$), and set difference(—).

The following subsection presents the algebraic laws manipulating the AND operator, while laws regarding the algebraic manipulations of regular relational operators are included in Appendix A.

### 3.1.1 Laws involving AND operator

Let $E_i$, for any $i$, stand for some relational expression; $F$ or $F_i$ for some condition; $A_i$, $B_i$, and $C_i$ for some attribute names.
1. Identity laws for AND operator.

If $F$ does not involve $E_2$,

$$\sigma_F(E_1 \times E_2) \equiv \sigma_F(E_1) \Lambda E_2 \quad (3.1)$$

If $F = F_1 \Lambda F_2$, $F_1$ involves $E_1$ and $E_2$ only, and $F_2$ involves $E_2$ and $E_3$ only,

$$\sigma_F(E_1 \times E_2 \times E_3) \equiv \sigma_{F_1}(E_1 \times E_2) \Lambda \sigma_{F_2}(E_2 \times E_3) \quad (3.2)$$

2. Commuting selection and projection with AND operator

If $F$ only involves $E_1$,

$$\sigma_F(E_1 \Lambda E_2) \equiv \sigma_F(E_1) \Lambda E_2 \quad (3.3)$$

If $F = F_1 \Lambda F_2$ and $F_1$ involves only $E_1$ and $F_2$ involves only $E_2$,

$$\sigma_F(E_1 \Lambda E_2) \equiv \sigma_{F_1}(E_1) \Lambda \sigma_{F_2}(E_2) \quad (3.4)$$

If $A_1, \ldots, A_n$ are attributes of $E_1$ but not of $E_2$,

$$\pi_{A_1, \ldots, A_n}(E_1 \Lambda E_2) \equiv \pi_{A_1, \ldots, A_n}(E_1) \Lambda E_2 \quad (3.5)$$

If $A_1, \ldots, A_n$ is a list of attributes of which $B_1, \ldots, B_m$ are attributes of $E_1$, and the remaining attributes, $C_1, \ldots, C_k$, are from $E_2$,

$$\pi_{A_1, \ldots, A_n}(E_1 \Lambda E_2) \equiv \pi_{B_1, \ldots, B_m}(E_1) \Lambda \pi_{C_1, \ldots, C_k}(E_2) \quad (3.6)$$

If $A_1, \ldots, A_n$ are attributes of both $E_1$ and $E_2$,

$$\pi_{A_1, \ldots, A_n}(E_1 \Lambda E_2) \equiv \pi_{A_1, \ldots, A_n}(E_1) \cap \pi_{A_1, \ldots, A_n}(E_2) \quad (3.7)$$
3. Associative law for AND operator.

\[(E_1 \land E_2) \land E_3 \equiv E_1 \land (E_2 \land E_3)\]  

(3.8)

### 3.1.2 Decomposition algorithm

The decomposition algorithm by algebraic manipulations of relational operators follows; equations 3.1 through 3.8 governing the manipulations of the new AND operator are listed in the preceding subsection, and equations A.1 through A.18 governing the manipulations of the existing relational operators are included in Appendix A.

**Input:** A relational expression represented as a relational operator tree.

**Output:** An optimized *decomposable* relational expression represented as a relational operator tree.

**Algorithm:**

1. Separate each selection \(\sigma_{F_1 \land \ldots \land F_n}(E)\) into the cascade \(\sigma_{F_1}((\ldots(\sigma_{F_n}(E))\ldots))\) using equations A.7 and A.8.

2. For each selection, move the selection as far down the tree as possible using equations A.10-A.14, A.16, and A.18.

3. Use the identity equations for AND operator, equations 3.1 and 3.2, to find independent expressions and introduce AND node into the tree. Then move the AND node as far up the tree as possible using equations 3.3-3.7. Multiple AND nodes can be consolidated into one with multiple child nodes as suggested in equation 3.8.
4. For each projection, move the projection as far down the tree as possible, using equations 3.5-3.7, A.10-A.11, A.15, and A.17.

5. Combine cascades of selections and projections into a single selection, a single projection, or a selection followed by a projection using equations A.7-A.11.

6. Group each child node of an AND or intersection operator and all its descendants into a subrule.

Example 5: The rule query R4 from Example 1 can be represented as the following relational expression from its definition, where a relation name preceded by \( \Delta \) identifies the triggering relation. This relational expression can also be represented as a relational operator tree as depicted in Figure 1. This expression is used as input to the above decomposition algorithm:

$$\pi_T (\sigma \quad I.name = "\text{DJIA}" \land \\
(\{I.current - I.open > 50\} \lor \{I.open - I.current > 50\}) \land \\
T.name = S.name \land \\
S.class = \text{major} \land \\
\{T.buyer = U.name\} \lor \\
\{T.seller = U.name\}) \land \\
U.class = \text{prog})$$ (3.9)

Step 1 transforms R4 by separating the selection into a cascade, which is presented in Figure 2.

$$\pi_T (\sigma \quad I.name = "\text{DJIA}" \land \\
(\{I.current - I.open > 50\} \lor \{I.open - I.current > 50\}) \land \\
T.name = S.name \land \\
S.class = \text{major} \land \\
\{T.buyer = U.name\} \lor \\
\{T.seller = U.name\}) \land \\
U.class = \text{prog}) (3.10)$$
Step 2 pushes the selections down the tree as close to the base relation it refers to as possible, which transforms R4 into the relational operator tree as in Figure 3.

\[ \pi_T(II \times SUT) \] \hspace{1cm} (3.11)

where II and SUT are

\[ II = \sigma_{T.name = "DJIA"} \sigma_{(I.current - I.open > 50) \lor (I.open - I.current > 50)} I \] \hspace{1cm} (3.12)

\[ SUT = \sigma_{T.name} \sigma_{(T.buyer = U.name)} ((\sigma_{S.class = major} S) \times (\sigma_{U.class = prog} U) \times \Delta T) \] \hspace{1cm} (3.13)

At step 3, R4 is transformed into the relational operator trees as in Figures 4 and 5, using equations 1 and 2 to identify possible replacement of Cartesian product with
Figure 2: Transformed Relational Operator Tree for Rule R4 After Step 1
Figure 3: Transformed Relational Operator Tree for Rule R4 After Step 2
Figure 4: Intermediate Relational Operator Tree for Rule R4 in Step 3
Figure 5: Transformed Relational Operator Tree for Rule R4 After Step 3
AND operators and equation 5 to push AND operators upwards in the following sequence:

\[
\pi_T(II \land STUT) \tag{3.14}
\]

\[
\pi_T(II \land ST \land UT) \tag{3.15}
\]

\[
II \land \pi_T(ST \land UT) \tag{3.16}
\]

where STUT, ST and UT are respectively

\[
STUT = \sigma_{T.name} \sigma_{(T.buyer = U.name)} ((\sigma_{S.class} S) \times \Delta T) \land (\sigma_{U.class} U) \times \Delta T \tag{3.17}
\]

\[
ST = \sigma_{T.name=S.name}((\sigma_{S.class=\text{major}} S) \times \Delta T) \tag{3.18}
\]

\[
UT = \sigma_{(T.buyer=U.name) \lor (T.seller=U.name)}((\sigma_{U.class=\text{prog}} U) \times \Delta T) \tag{3.19}
\]

At step 4, equation 7 is used to push the projection operator down, which transforms R4 into the relational operator tree as in Figure 6,

\[
II \land (PST \cap PUT) \tag{3.20}
\]

where PST and PUT are

\[
PST = \pi_T(\sigma_{T.name=S.name}(\sigma_{S.class=\text{major}} S) \times \Delta T)) \tag{3.21}
\]

\[
PUT = \pi_T(\sigma_{(T.buyer=U.name) \lor (T.seller=U.name)}((\sigma_{U.class=\text{prog}} U) \times \Delta T)) \tag{3.22}
\]

At step 4, equation 7 is used to push the projection operator down, which transforms R4 into the relational operator tree as in Figure 6, which can also be represented as the following subrules:
$\sigma(\text{name}="\text{DJIA}" \land (\text{current} - \text{open} > 50) \lor (\text{open} - \text{current} > 50)) I$  \hspace{1cm} (R4_1)

$\pi_T(\sigma_{\text{T.name}} = \text{S.name} ((\sigma_{\text{S.class}=\text{major}} S) \times \Delta T))$  \hspace{1cm} (R4_2_1)

$\pi_T(\sigma_{(\text{T.buyer}=\text{U.name}) \lor (\text{T.seller}=\text{U.name})} ((\sigma_{\text{U.class}=\text{prog}} U) \times \Delta T))$  \hspace{1cm} (R4_2_2)

$R4_2_1 \land R4_2_2 \hspace{1cm} (R4_2)$

$R4_1 \land R4_2 \hspace{1cm} (R4)$
Example 6: Likewise, rules R1, R2, and R3 from Example 1 can be transformed by the decomposition algorithm. R1 is transformed into one subrule

\[ \pi_{S.current}(\sigma_{S.class=major} \Delta S) \]  

(R1)

R2 is transformed into one subrule

\[ \sigma_{(I.open-I.current>100) \land (I.name="DJIA")} \Delta I \]  

(R2)

R3 is transformed into

\[ \pi_{S.high,S.current}(\sigma_{S.current>S.high} \Delta S) \]  

(R3)

3.2 Rule Distribution

Assume a base relation is partitioned into some data segments horizontally, vertically, or a combination of both, and each data segment is distributed to a different site. A horizontal partition is identified by a guard selection, which must be satisfied by every tuple in the partition; while a vertical partition is identified by a guard projection, which contains a subset of the attributes of the original relation, and is shared by all tuples in the partition. The distributed representation of the original base relation is a union of the horizontal partitions, a natural join of the vertical partitions, or a combination of both.

Each subrule as derived from the previous decomposition algorithm can be further decomposed for distribution as follows. Replace each reference of the base relation in the subrule with its distributed representation. This replaced subrule can then be
transformed by the same decomposition algorithm presented in the previous section to push selections and projections present in the original subrule down through the newly introduced union and join operators and their associated guard selections and projections. When combining an existing selection with the guard selection of some data segment, an entire subtree might be deleted due to contradiction of the two selection conditions. When combining an existing projection with the guard projection of some data segment, an entire subtree might be deleted due to the empty intersection between the two sets of attributes. After this transformation, each subrule can be viewed as a tree with the union or join operator at the root with some subtrees as children. Each of these subtrees can then be distributed to the site where the corresponding data segment, identified by the guard selection or projection, is allocated.

Example 7: Assume the STOCKS relation from Example 1 is distributed to 10 sites according to possible values of the district attribute. Then the distribution algorithm will distribute subrule $R3_1$ to these 10 sites, each as a subrule of

$$\pi_{S.high,S.current}(\sigma(S.current > S.high) \land (district=\ldots) \Delta S)$$

(R3$_1$)

Likewise, subrule $R4_{21}$ will be transformed into

$$\pi_T(\sigma_{T.name=S.name}((\cup R4_{21}) \times \Delta T))$$

(R4$_{21}$)

$$\sigma(S.class=\text{major}) \land (district=\ldots) S$$

(R4$_{21}$)

$\square$
4.1 Basic Ideas

A distributed rule query is modeled as a relational expression tree, where leaf nodes are allocated on different sites and an internal node might be allocated on the same site as one of its children. In general, all nodes of a tree calculate some relational expressions and pass the results to their parents; specifically, a leaf node derives its relation from some local base relation, while an internal node derives its relation from relations received from its children. The relation derived by the root node represents the binding between the rule condition and the rule action. When this relation is not empty, the root node is responsible for alerting the system and forwarding this data to the rule action. The evaluation algorithm in a distributed environment determines when a rule query is true in a timely and consistent way with minimum communication overhead.

To evaluate subrules scattered among nodes in a distributed environment where no global clock and shared memory are available and many updates may happen simultaneously, coordination among nodes is necessary to guarantee the correct evaluation
result. The *simultaneous region* approach [60] offers such a coordination scheme for a
tree of nodes. Execution at each node is divided into regions, each of which is iden-
tified by a region number. Regions with the same number among a group of sibling
nodes are in effect in execution simultaneously and, thus, form a simultaneous region.

When a local update occurs at a node, the node will initiate a simultaneous region
evaluation to propagate the change incurred by the update upwards. A node initiates
a simultaneous region evaluation by sending its siblings *synchronization*(sync) mes-
sages, which contain nothing but its own region number, before advancing its own
region. Upon receiving a sync message, a node will advance its region if it has not
done so already due to its own local update. Whenever a node advances its region, it
sends to its parent the latest evaluation result of the local relational expression with
the region number in a *data* message before it increments its region number by 1.

An internal node will derive its relation with children relations from a simultaneous
region, namely, with relations received from its children in *data* messages with the
same region number. The new evaluation result is considered a local update to this
internal node and triggers a simultaneous region evaluation among its siblings. Hence,
an internal node maintains two separate region numbers, one for its child nodes and
the other for its sibling nodes. The various messages to propagate an update from a
triggering leaf node to the root node is depicted in Figure 7.

### 4.2 Procedure Pseudocodes

Assume a reliable underlying network which delivers messages in the order sent and
loses no messages. Also assume at each site *current.region.no* is initialized to 0
and \textit{state} is initialized to reflect the proper subrule state. The pseudocodes of the procedures to be executed on nodes of a distributed rule query tree are as follows. Procedure \textit{local\_update()} and \textit{recv\_sync()} are to be executed on all nodes, while procedure \textit{recv\_data()} is to be executed on internal nodes only.

\textbf{Procedure local\_update}

\/* Execute this procedure when a local update occurs */

Begin
send sync message to all sibling nodes with current-region-no;

state-info := eval(local expression);

send data message to parent node with state-info, site-id, and current-region-no;

current-region-no := current-region-no + 1;

End

Procedure recv-sync(sibling-region-no)

/* Execute this procedure when a sync message from a sibling node is received */

Begin

while current-region-no != sibling-region-no then

Begin

send data message to parent node with state-info, site-id, and current-region-no;

current-region-no := current-region-no + 1;

End

End

Procedure recv-data(child-state-info, child-id, child-region-no)

/* Execute this procedure when a data message from a child node is received */

Begin

store child-state-info in a two-dimensional table indexed by child-id and child-region-no;

if child-state-info of all child nodes are available for the child-region-no then

Begin
if parent = Null then

    Begin

    state_info := eval(child_state_info of all child nodes for the child_region_no);

    if state_info.status = True then

        Begin

        recognize the occurrence of rule condition;

        pass state_info to rule action;

        End

        else nop;

        End

    else call procedure local_update;

    release table space for all child_state_info with region number \leq child_region_no;

    End

End

Examples to demonstrate the operation of this distributed rule evaluation algorithm are presented next.

4.3 Sample Evolutions of the Distributed Rule Evaluation

To demonstrate the operation of the distributed rule evaluation algorithm, the distributed rule tree shown in Figure 8 will be used as the basis for the examples. Node 0 is the root node, which is responsible for reporting to the system when the entire rule query is changed from True to False and from False to True. Assume that Node
Figure 8: Sample Distributed Rule Tree

0 is true when both Node 1 and 2 are true, and Node 2 is true when Nodes 21, 22, and 23 are all true. The schematic diagram in Figure 10 will be used to demonstrate the basic operation of the distributed rule evaluation algorithm. Legends used in Figure 10 are explained in Figure 9.

Example 8: [basic operation] Assume that Nodes 1 and 21 are True and the rest of the nodes are False at present time. Also assume that Node 0 is in region 20, Nodes 1 and 2 are in region 15, and Nodes 21, 22, and 23 are in region 10. Further assume that a local update occurs at Node 22, changing its status to True. The events following this local update are illustrated in the top half of Figure 10.

As soon as Node 22 detects the local update, sync messages are sent to its siblings, Nodes 21 and 23, with region number 10. Node 22 recomputes the local relational expression taking into account the new update. The result from the recomputation is sent to its parent, Node 2, in a data message with the True status and region number 10. After the data message is sent, Node 22 increments its region number to 11.
When Node 23 receives the sync message with region number 10 from Node 22, it sends its current status without recalculation to its parent, Node 2, since no local update occurs during this region. Then Node 23 increments its region number to 11. Node 21 will act similarly.

When Node 2 receives all three data messages with region number 10 from its child nodes, it sends to its sibling, Node 1, a sync message with region number 15, and recalculates its local expression. Since Node 23 is still False, Node 2's status remains False. The newly calculated result is then sent to Node 0 in a data message with the False status and region 15 before the region number is incremented to 16.

When Node 1 receives the data message with region number 15 from its sibling, it sends its current status to Node 0 since no local update occurs in this region. The data message will bear the current True status and region number 15. After the data message is sent, Node 1 increments its region number to 16.

When Node 0 receives both data messages from Nodes 1 and 2 for region 15, it recalculates the local expression. Since Node 2 is still False, the new result for Node 0 is assumed to remain False. No further action is taken, except that the local region number is incremented to 21.

Example 9: [concurrent updates at siblings nodes] Continued from the previous example, this example demonstrates what might happen when multiple updates occur concurrently at sibling nodes. Assume that local updates occur at Nodes 22 and 23 simultaneously and both updates toggle the local status. The events following the occurrence of these two updates are illustrated in the bottom half of Figure 10.
Legend

- -- local update
- -- data message
- -- sync message
- -- region number changed

n/S -- node status, where
  n is local region no.,
  S is local state,

<n> -- content of sync msg,
  n is region no.

<S,m,n> -- content of data msg,
  S is node state,
  m is node no.,
  n is region no.

Figure 9: Legends for Evolution of the Distributed Rule Evaluation

Upon the detection of the local update, Node 22 sends a sync message to its siblings with region number 11. It recalculates its local expression, which is changed from True to False. The new status is sent to Node 2 in a data message with region number 11, before the local region number is incremented to 12. Node 23 will do likewise. Upon receipt of the first sync message with region number 11, Node 21 sends a data message and increments the local region number as in the previous example.

When Node 2, which is in region 16, receives all three data messages for child region 11, it sends a sync message to Node 1 and recalculates the local expression. Since both local updates at Nodes 22 and 23 are captured in the data messages for child region 11, their effects will be combined together by Node 2 in the recalculation. The new result will be reported to its parent, Node 0, in a data message with region
Figure 10: Evolution of the Distributed Rule Evaluation
Figure 11: Evolution of the Distributed Rule Evaluation

number 16. When Node 0 receives both data messages for region 16, it will recalculate to determine the status of the entire rule query.

When Node 22, in region 12, receives the second sync message with region number 11, it will do nothing since its status for region 11 has already been reported to its parent node. Node 23 ignores the sync message with region number 11 from Node 22 as well; so does Node 21 with the second sync message with region number 11. □
Example 10: [skipping region number] Continued from the previous example, this example demonstrates a possible situation where a node might receive a sync message with a region number larger than its own local region number. Assume a local update occurs at Node 22 first and then, without much delay, another local update occurs at Node 23. Further assume the local update at Node 23 occurs after Node 23 reports its status to the parent node for the first update. A possible sequence of events following the occurrence of these two updates are illustrated in Figure 11.

Upon the detection of the first update, Node 22 sends sync messages with region number 12 to Node 23 and 21, recalculates its local expression, sends the new True status to Node 2 in a data message with region number 12, and increments its local region number to 13. When Node 23 receives the sync message with region number 12 from Node 22, it sends its current status without recalculation to Node 2 since no local update has occurred in this region. Later, when the second update occurs at Node 23, it sends sync message with region number 13 to Nodes 22 and 21, recalculates the local result, sends the new False status to Node 2 in a data message with region number 13, and increments its local region number to 14.

Due to communication delay, assume Node 21 receives the sync message with region number 13 from Node 23 before it receives the sync message with region number 12 from Node 22. Since Node 21 is still in region 12, it sends two consecutive data messages with region numbers 12 and 13 to its parent and updates its local region number accordingly. Because no local update occurs in this region, the current status without recalculation is reported in both data messages. When Node 21 finally
receives the sync message with region number 12 from Node 22, it just throws away the message since it has already reported it status to its parent for region number 12.

\[\square\]

**Example 11:** [concurrent updates at different levels] Continued from the previous example, this example demonstrates a possible evolution when concurrent updates occur at different levels. Assume that two updates occur concurrently at Node 1 and Node 23. The events following the occurrence of these two updates are depicted in Figure 12.
Upon the detection of the local update, Node 23 sends sync messages with region number 14 to Nodes 21 and 22, recalculates its local expression, sends the new True status to Node 2 in a data message with region number 14. Upon receiving the sync messages numbered 14 from Node 23, both Node 21 and Node 22 send the current status to Node 2 in data messages with region number 14.

When Node 1 detects the local update, it sends a sync message with region number 19 to Node 2, recalculates its local expression, sends the new False status to Node 0, and increments the local region number to 20.

Assume that Node 2 receives the incoming messages in the following order: the data message with region number 14 from Node 22, the data message with region number 14 from Node 21, the sync message with region number 19 from Node 1, and the data message with region number 14 from Node 23. As soon as Node 2 receives the sync message from Node 1, it needs to report its status to Node 0. Although Node 2 has already received two of the data messages for child region 14, it cannot recalculate its local expression due to the missing data message from Node 23. Node 2 can only resend its status from the last recalculation to Node 0 in a data message with region number 19 before it increments the local region number to 20. When Node 20 finally receives the data message with region number 14 from Node 23, it recalculates its local expression with the data collected from all three data messages with region number 14 and increments its local region number to 21. Before the recalculation, Node 2 sends a sync message numbered 20 to Node 1 to request a synchronized status update.
Assume that before Node 1 receives the sync message with region number 20 from Node 2, a local update occurs. In response, Node 1 sends a sync message with region number 20 to Node 2, recalculates its local expressions, sends the new result to Node 0 in a data message with region number 20, and finally increments its region number to 21. When Node 1 in region 21 receives the sync message with region number 20 from Node 2, it ignores the sync message since it has already reported its status for region 20. For the same reason, Node 2 will ignore the sync message with region number 20 from Node 1 when it finally arrives. □

4.4 Correctness Proof

Assume that

- the underlying communication network is reliable and loses no messages;
- the channels between any two nodes deliver messages in the First In First Out (FIFO) order;
- the three procedures, local_update(), recv_sync(), and recv_data(), are executed as atomic processes; and
- current_region_no at each node is initialized to 0.

Then the distributed rule evaluation algorithm among a group of $m$ sibling nodes, where $m \geq 2$, with a common parent node has the following properties.
Property 1: No region number will be skipped at any node.

`current_region_no` is only incremented by 1 every time it is updated in `recv_sync()` or `local_update()`. Since `current_region_no` is initialized to 0 at every node and the processes are atomic, no region number will be skipped at any node.

Property 2: Regions at any node are successive. That is, if regions for node \( r \) are represented by \( r_{\text{current\_region\_no}} \), and the node is currently in region \( i \), then regions for node \( r \) are ordered \( r_0, r_1, \ldots, r_{i-1}, r_i \).

Assume the `current_region_no` for a node \( r \) is \( i \). The region number can change in only two ways: as a result of a local update or as the result of receiving a sync message. If an update occurs to the node locally, the process `local_update()` will be executed and `current_region_no` will be incremented by 1. Thus, the region number will change from \( i \) to \( i + 1 \). If a sync message is received, `recv_sync()` will be executed and `current_region_no` will be incremented according to the relationship between the received sibling region number and `current_region_no`. Three cases are possible. Each of the three cases is discussed below in detail.

**Sibling region number = current region number:** Assume node \( r \) in region \( i \) receives a sync message numbered \( i \) from node \( s \). This situation can occur only when a local update occurs at node \( s \) while no local update occurs at node \( r \) since the last region number increment at node \( r \). In this case, `current_region_no` will change from \( i \) to \( i + 1 \), which will terminate the while loop. An example situation is demonstrated in Figure 13.
Figure 13: sibling_region_no = current_region_no

Figure 14: sibling_region_no > current_region_no
sibling_region_no > current_region_no: When in region $i$, node $r$ may receive a sync message numbered $j$ from node $s$, where $i < j$. This situation may arise when there are nodes $p_i, p_{i+1}, ..., p_{j-1}$ that have local updates occurring at their own sites and have sent sync messages numbered $i, i + 1, ..., j - 1$, where $\forall i \leq i < j$ $p_i \neq r$ and $p_i \neq s$. Node $s$ has received the sync message numbered $j - 1$ before its own local update occurs and node $r$ has not received sync messages numbered from $i$ to $j - 1$ before it receives the sync message numbered $j$ from node $s$. So when node $r$ receives the sync message numbered $j$, it needs to bring itself up to date to region $j$ and process this request as described in the case $sibling\_region\_no = current\_region\_no$. Inside the while loop of $recv\_sync()$, $current\_region\_no$ is incremented by 1 repeatedly until $current\_region\_no > j$. An example situation is demonstrated in Figure 14.

If any node $p_i$ from nodes $p_i, p_{i+1}, ..., p_{j-1}$ is node $s$, then node $s$ has sent a sync message numbered $l$, where $i \leq l < j$ prior to sending the sync message numbered $j$. However, because the underlying network is assumed to lose no messages and delivers messages in the order sent, node $r$ could not have received the sync message with the larger region number without receiving the sync message sent previously by the same node.

sibling_region_no < current_region_no: Assume node $r$ in region $i$ receives a sync message numbered $j$, where $j < i$, from node $s$. Since node $r$ is already in region $i$, $i > j$, and, from Property 1, no region number is skipped at any node, node $r$ has incremented region number from $j$ to $j + 1$ because a local update has occurred or because a sync message has been received from node $p$, where $p \neq s$, with region
Figure 15: sibling_region_no < current_region_no
number $k$, where $j \leq k < i$. If a local update has occurred, a sync message numbered $j$ has been sent from node $r$ to node $s$. However, since node $s$ has sent a sync message numbered $j$, node $s$ did not receive the sync message $j$ from node $r$ before the update at node $s$. The two sync messages numbered $j$ from node $r$ and $s$ have crossed paths. A sample situation is demonstrated in Figure 15(a).

Due to the computation invoked by the local update to recalculate local status, a delay will be introduced between the sending of the sync message and the actual increment of the region number at node $r$. A sync message from other node may arrive at node $r$ during this delay. However, because the processes are assumed to be atomic, the arrived sync message will not be read by the process recv_sync() until process local_update() has finished. The situation depicted in Figure 15(c) will actually be processed like the situation demonstrated in Figure 15(d).

If node $r$ has incremented region number $j$ because a sync message has been received from another node, current_region_no was incremented when that sync message was received as described in the case of \texttt{sibling_region_no = current_region_no} or \texttt{sibling_region_no > current_region_no}. A sample situation is demonstrated in Figure 15(b).

Whether node $r$ incremented region $j$ due to a local update or due to a sync message from another node, no action needs to be taken now. The test condition for the while loop in recv_sync() is not satisfied and current_region_no is not changed. □

**Property 3:** For every region in a node, there is a corresponding, equally numbered region in its sibling nodes.
Assume the last region number at node $r$ is $i$. We want to prove that region $i$ will exist, if it has not, in any of the sibling nodes, say node $s$. There are only three possible ways for node $r$ to reach region $i$, the occurrence of its own local update, the receipt of sync message from node $s$ for region $i$, and the receipt of sync message from another node for region $i$. If node $r$ reaches region $i$ due to its own local update, node $r$ has sent a sync message to node $s$. With the assumption that the network loses no message, node $s$ would eventually enter region $i$ due to the receipt of this sync message or due to its own local update. If node $r$ reaches region $i$ due to the receipt of sync message from node $s$, then node $s$ must have been in region $i$ already. If node $r$ reaches region $i$ due to the receipt of a sync message from another node, say node $p$, then node $p$ would have sent an equally numbered sync message to node $s$. Thus node $s$ would eventually enter region $i$ due to the receipt of this sync message or its own local update.

Property 4: Data messages received from a child node carry successive region numbers.

A data message is sent whenever current_region_no is incremented by 1 in local_update() or in recv_sync(). Since a data message is sent for each distinct value of current_region_no and from Property 2, regions are successive at each node, data messages carrying successive region numbers will be sent from a node to its parent. With the assumption that no message is lost and messages arrive in the order sent, data messages received from a child node will carry successive region numbers.
Property 5: A simultaneous region will be formed at the parent node by receiving data messages with the same region number from every child node.

From Property 4, data messages received from a child node carry successive region numbers. For each \( i \) of these region numbers, there is a corresponding, equally numbered region in its sibling nodes from Property 3. A data message will be sent with region number \( i \) from each node \( s \) of its siblings. Thus the parent node would eventually receive data messages with region number \( i \) from every one of its child nodes and, thus, form a simultaneous region with region number \( i \).

Based on these properties, we can show that the distributed rule evaluation algorithm is correct, that is, it exhibits no erroneous behavior. A distributed rule evaluation algorithm behaves erroneously if either of two conditions arise:

- a rule condition is true but is not recognized; and
- a rule condition is false but is recognized as true.

The assertion that the distributed rule evaluation algorithm exhibits no erroneous behavior will be examined separately in the two claims below.

Claim 1: Every true status of a rule query will be recognized.

Each update will be captured in a region by the local node and a data message carrying the effect of this update will be sent to the parent node with this region number. From Property 5 a simultaneous region with this region number will be formed at the parent node by this data message and data messages carrying the same region number from its siblings. All data messages for this simultaneous region will be combined to derive the parent node’s latest status. This derivation is considered
a local update at this parent node and will initiate another simultaneous region advancement at this level. Thus, if the subrule query becomes true at this parent node, the true status will be transmitted up the rule query tree to the root. If the status of the root node is true in a simultaneous region, the status of the entire rule query is true and will be recognized. □

Claim 2: No false status of a rule query will be recognized as true.

A parent node will derive its local result with data messages from a simultaneous region only. If the local status is false, the parent node will transmit the false status up the tree to the root node. If a rule query is not true in a simultaneous region at the root node, the root node will not recognize it as true status. Therefore, no false recognition can be made. □
CHAPTER V

PERFORMANCE STUDY OF THE DISTRIBUTED RULE EVALUATION ALGORITHM USING SIMULTANEOUS REGIONS

Among the three algorithms for monitoring rules in a distributed active database, the rule decomposition and distribution algorithms are executed at rule definition time, whereas the distributed rule evaluation algorithm is executed constantly as long as the database is operational. For databases whose rules are not changed very often, the performance of the distributed rule evaluation algorithm plays the key role in the overall performance of the distributed active databases. An analytical performance model has been developed to study the performance of the distributed rule evaluation algorithm presented in the previous chapter. From the analytical model we derive the number of messages and the mean response time to propagate a triggering update, which toggles the global status of a rule condition. The total message count in support of the operation of this distributed rule evaluation algorithm, however, is not easy to derive analytically due to the simultaneous region approach adopted in the distributed rule evaluation algorithm, where multiple messages may be combined by an ancestor node. The total message count is collected from a simulation instead. To
verify the correctness of the simulation, the mean response time collected from the simulation is compared with the analytical data, which confirms the correctness of both the analytical and simulation models.

5.1 Notations and Definitions

To study the performance of the distributed rule evaluation algorithm, a distributed rule query is modeled as a tree of nodes. Database updates may occur to any of the leaf nodes. The very last database update that makes the global status of the rule query change from false to true or from true to false is termed the triggering update. The node where the triggering update occurs is termed the triggering leaf node. After the triggering leaf node recognizes the occurrence of a triggering update, it initiates a simultaneous region evaluation among itself and its siblings in order to propagate the update upwards. The parent node of the triggering leaf node, after a consistent set of evaluation results is collected from all its children, will recognize the occurrence of the triggering update and become the triggering node at its level. The parent node has the responsibility to initiate a simultaneous region evaluation among its siblings to make the triggering update known to its parent. Eventually, the root node will recognize the occurrence of the triggering update and the changing status of the rule query.

Definitions: The level of a node is the number of edges in the path from the node to the root node. Hence, the root node has a level 0. The direct ancestors of a node are all the nodes along the path from that node to the root. The ancestors of a node are all the nodes whose parents are direct ancestors of the node and whose levels are
Figure 16: Sample Model of Distributed Rule

less than that of the node. The siblings of a node are all the nodes who share the same parent with it.

Example 12: Figure 16 presents a sample tree of 12 nodes. Node J is at level 3. Nodes F, B and A are direct ancestors of J. Nodes E, F, B, C, D, and A are ancestors of J. Nodes K and P are J’s siblings.

For a triggering update to propagate to the root node, the siblings and the ancestors of the triggering leaf node are all involved in the evaluation process. Assume the triggering leaf node is at level \( \ell \) with \( m - 1 \) sibling nodes and \( n \) ancestor nodes. The message count and the response time for the triggering update to propagate to the root node are analyzed in the following sections.

5.2 Derivation of Message Count

The number of messages exchanged for the triggering update to propagate to the root node from the triggering leaf node can be analyzed in terms of two types of
messages, sync messages and data messages. Sync messages are fixed-sized, short, and are exchanged between sibling nodes. Data messages have varying sizes, have potential to be long and are passed from children to parent node.

For a triggering update to propagate one level up from a node with \( m - 1 \) siblings, \( m - 1 \) sync messages and \( m \) data messages will be sent out for a simultaneous region evaluation at that level. The number of data messages sent is one plus the number of sync messages sent, because the triggering node does not send a sync message to itself.

For a triggering update originating at level \( \ell \) to propagate up to the root, the above simultaneous region evaluation will be repeated \( \ell \) times. Every sibling and every ancestor of the triggering leaf node will participate in this evaluation process. Every participating node except the root will send out a data message. Thus, the total number of data messages sent for the triggering update to propagate to the root is equal to the total number of ancestor nodes of the triggering leaf node minus 1 for the root node plus the total number of sibling nodes of the triggering leaf node plus 1 for itself. This number is \( n - 1 + m \).

At each level, the number of sync messages required is one less the number of data messages required. Hence, the total number of sync messages required for the triggering update to reach the root node is equal to the total number of data messages minus the level of the triggering leaf node. This number is \( n - 1 + m - \ell \).
5.3 Derivation of Response Time

Response time will be measured as the time elapsed from the moment the triggering leaf node recognizes the occurrence of the triggering update to the moment the root node recognizes the changed status of the entire rule query. Updates are assumed to arrive at a node according to Poisson distribution [17] and the mean update arrival rate among sibling nodes is identical.

Let random variable $Z$ denote the time elapsed from the moment the triggering node recognizes the occurrence of the triggering update to the moment it finishes the local calculation incurred by the update. Assume this computation delay has an Exponential distribution with means $\frac{1}{\lambda}$. Let random variable $Y$ denote the time elapsed from the moment the triggering node sends out the data message to the moment the parent node receives the data message. Assume this communication delay has an Exponential distribution with means $\frac{1}{\lambda}$. From definition, the probability density functions (pdf's) for these two random variables are:

$$f_Y(y) = \begin{cases} \lambda e^{-\lambda y} & y \geq 0 \\ 0 & y < 0 \end{cases}$$  \hspace{1cm} (5.1)

$$f_Z(z) = \begin{cases} \lambda' e^{-\lambda' z} & z \geq 0 \\ 0 & z < 0 \end{cases}$$  \hspace{1cm} (5.2)

Let random variable $V$ denote the time elapsed from the moment the triggering node recognizes the occurrence of the triggering update to the moment the parent node receives the data message from the triggering node. Then $V = Y + Z$. Let $f_{YZ}(y, z)$ denote the joint pdf for $Y$ and $Z$. Assuming $Y$ and $Z$ are independently
distributed, \( f_{YZ}(y, z) = f_Y(y)f_Z(z) \) [17]. The pdf for \( V, g(v) \), can be derived from \( f_{YZ}(y, z) \). Note that \( V \leq v \) whenever \( Y \leq y \) and \( Z \leq v - y \). Thus,

\[
g(v) = \int_0^v f_{YZ}(y, v - y)\,dy
\]

\[
= \int_0^v f_Y(y)f_Z(v - y)\,dy
\]

\[
= \int_0^v \lambda e^{-\lambda y}e^{-\lambda'(v - y)}\,dy
\]

\[
= \lambda\lambda' e^{-\lambda y} \int_0^v e^{-(\lambda - \lambda')y}\,dy
\]

\[
= \frac{\lambda\lambda'}{\lambda - \lambda'} e^{-\lambda y} \int_0^v e^{-(\lambda - \lambda')y}\,dy
\]

\[
= \frac{\lambda\lambda'}{\lambda - \lambda'} (e^{-\lambda y} - e^{-\lambda' y})
\]

(5.3)

From the pdf \( g(v) \), the cumulative distribution function (cdf) for \( V, G(v) \), can be derived.

\[
G(v) = \int_0^v g(t)\,dt
\]

\[
= \int_0^v \frac{\lambda\lambda'}{\lambda - \lambda'} (e^{-\lambda t} - e^{-\lambda' t})\,dt
\]

\[
= \frac{\lambda\lambda'}{\lambda - \lambda'} \left[ \int_0^v e^{-\lambda' t}\,dt - \int_0^v e^{-\lambda t}\,dt \right]
\]

\[
= \frac{\lambda\lambda'}{\lambda - \lambda'} \left[ \frac{1}{\lambda'} (e^{-\lambda' v} - 1) - \frac{1}{\lambda} (e^{-\lambda y} - 1) \right]
\]

\[
= \frac{\lambda'}{\lambda - \lambda'} (e^{-\lambda y} - 1) - \frac{\lambda}{\lambda - \lambda'} (e^{-\lambda' y} - 1)
\]

(5.4)

The pdf and cdf of the time delay for the parent node to receive the data message from a non-triggering node can be similarly derived. Let random variable \( X \) denote the communication delay for the sync message. Let random variable \( U \) denote the time elapsed for the parent node to receive the data message corresponding to the update.
from a sibling of the triggering node. Then \( U = X + Y \). Assume \( X \) has Exponential distribution with the same mean \( \frac{1}{\lambda} \) as \( Y \), and \( X \) and \( Y \) are independently distributed.

Let \( f \) denote the pdf and \( F \) the cdf for \( U \).

\[
\begin{align*}
f(u) &= \int_0^u f_{XY}(x, u-x)dx \\
&= \int_0^u f_X(x)f_Y(u-x)dx \\
&= \int_0^u \lambda e^{-\lambda x} \lambda e^{-\lambda(u-x)}dx \\
&= \lambda^2 e^{-\lambda u} \int_0^u 1dx \\
&= \lambda^2 u e^{-\lambda u} 
\end{align*}
\]

(5.5)

\[
\begin{align*}
F(u) &= \int_0^u f(t)dt \\
&= \int_0^u \lambda^2 t e^{-\lambda t}dt \\
&= \lambda^2 \int_0^u t e^{-\lambda t}dt \\
&= \lambda^2 \left[ \frac{t}{\lambda} e^{-\lambda t} \bigg|_0^u - \int_0^u \frac{1}{\lambda} e^{-\lambda t}dt \right] \\
&= -\lambda u e^{-\lambda u} + \lambda \int_0^u e^{-\lambda t}dt \\
&= -\lambda u e^{-\lambda u} + \lambda \left[ -\frac{1}{\lambda} e^{-\lambda t} \bigg|_0^u \right] \\
&= -\lambda u e^{-\lambda u} - e^{-\lambda u} \bigg|_0^u \\
&= 1 - e^{-\lambda u} - \lambda u e^{-\lambda u} 
\end{align*}
\]

(5.6)

For a node with \( m \) children, the time elapsed to pass the triggering update one level up to it is the time delay for this node to receive the last data message for the simultaneous region incurred by this update from its children. Let random variable \( T \) denote this time delay. Let random variables \( S_i, 1 \leq i \leq m \), denote the time delay for this node to receive the data message from child node \( i \). Without loss
of generality, assume child node 1 is the triggering child node. $S_1$ will have a cdf of $G$; while $S_2, \ldots, S_m$ will have cdf of $F$. Assume $S_t$'s are mutually independent.

The probability for the parent node to receive the last data message incurred by the triggering update before $t$ units of time is for it to receive a data message from every one of its child nodes before $t$ units of time, i.e.,

$$F_T(t) = \prod_{i=1}^{m} F_{S_i}(t)$$

$$= G(t) F(t)^{m-1} \quad (5.7)$$

Then the pdf for $T$ can be derived.

$$f_T(t) = \frac{\partial F_T(t)}{\partial t}$$

$$= \frac{\partial G(t) F(t)^{m-1}}{\partial t}$$

$$= (m-1) F(t)^{m-2} \frac{\partial F(t)}{\partial t} G(t) + F(t)^{m-1} \frac{\partial G(t)}{\partial t}$$

$$= (m-1) f(t) F(t)^{m-2} G(t) + g(t) F(t)^{m-1} \quad (5.8)$$

When the mean computation delay is the same as the mean communication delay, $\lambda = \lambda'$, the pdf and the cdf of the delay experienced by the parent node in receiving a data message from either the triggering node or a sibling of the triggering node are as follows.

$$g(v) = f(u) = \lambda^2 u e^{-\lambda u} \quad (5.9)$$

$$G(v) = F(u) = 1 - e^{-\lambda u} - \lambda u e^{-\lambda u} \quad (5.10)$$
The pdf and cdf of the response time delay experienced by the parent node in recognizing the triggering update which occurs in one of its \( m \) child nodes are as follows.

\[
F_T(t) = F(t)^m \quad (5.11)
\]

\[
f_T(t) = mF(t)^{m-1}f(t) \quad (5.12)
\]

For any given \( \lambda, \lambda', \) and \( m \), the mean response time to propagate an update one level up is the expected value of \( f_T(t) \), \( E \).

\[
E = \int_0^\infty t f_T(t) dt \quad (5.13)
\]

For a triggering node at level \( \ell \), let \( E^i \) denote the mean response time to propagate the update one level up from level \( i \) to level \( i - 1 \). The mean response time to propagate the update from the triggering leaf node up to the root node is then \( \sum_{i=1}^\ell E^i \).

### 5.4 Numerical Results

Since the integration in equation 5.13 does not have a closed form, the expected value of \( f_T(t) \) is computed numerically using the following formula, which is also presented graphically in Figure 17.

\[
E = \sum_{n=0}^{Tlength} \frac{n(\Delta T)^2 f_T(n\Delta T)}{\Delta T} \quad (5.14)
\]

\( \Delta T \) is the width of the increment along the \( T \) axis and should be narrow enough in order to minimize the error around the peak of \( f_T(t) \). \( T\text{length} \), chosen to terminate the summation within an acceptable error range, is the maximal value that will be
Figure 17: Computation Model for Calculating Mean Response Time Numerically
used to calculate $f_T(t)$ since $f_T(t)$ approaches zero asymptotically after the peak. In our numerical computation, we vary $\Delta T$ and $T_{length}$ from 0.01 to 0.001 and from 0.9 to 4.2, respectively, to achieve an error margin of 1% in calculating the area under the curve, which is 1.

The mean response time is computed numerically with parameters $\lambda$, $\lambda'$, and $m$. Parameter $\lambda$ is the inverse of the mean communication delay for one message transmitted over the network. Parameter $\lambda'$ is the inverse of the mean computation delay in calculating a relational expression. Parameter $m$ is the number of sibling nodes plus one for the triggering node.

The mean response time to propagate a triggering update one level up from a group of $m$ nodes is plotted with varying parameter values. Figure 18 plots the mean response time with mean computation delay kept at 100 msec and mean communication delay ranging from 10 msec to 60 msec. Figure 19 plots the mean response time with mean communication delay kept at 20 msec while mean computation delay ranging from 100 msec to 500 msec. Figure 20 plots the mean response time with equal mean communication and computation delay which ranges from 5 msec to 40 msec.

Figures 18, 19, and 20 show that the mean response time is a monotonically increasing function with decreasing slopes with the number of child nodes. Let $E_m$ denote the mean response time to propagate an update up a level of $m$ nodes. The following inequalities are observed for any $m > 2$:

$$E_m < E_{m-1} + E_2$$  \hfill (5.15)
Mean Comp Time = 0.1 Sec

Mean response time (sec) x 10^-3

Figure 18: Analytical Mean Response Time for Various Mean Communication Delays
Comm Delay = 0.02 Sec

Mean response time (sec) x 10^{-3}

Figure 19: Analytical Mean Response Time for Various Mean Computation Delays
Figure 20: Analytical Mean Response Time When the Mean Computation Delay is Equal to the Mean Communication Delay
The right hand sides of the inequalities represent the mean response times to propagate an update two or more levels up from trees which correspond to various rearrangements of the original \( m \) nodes. These inequalities indicate that the mean response time to propagate an update one level up is always smaller than the mean response time to propagate the update two or more levels up for any arrangement of the same number of leaf nodes.

### 5.5 Simulation Data for Mean Response Time

The distributed rule evaluation algorithm is also simulated using SIMPAS, an event-driven discrete system simulation language based on PASCAL. Response times observed from a sufficiently large number of regions were averaged to derive the simulated response time.

Assume the mean update arrival rate at any child node is 0.01/sec. Figure 21 plots the mean response time to propagate an update from a group of \( m \) nodes to the parent node, with the mean computation delay kept at 100 msec and the mean communication delay varying from 10 msec to 50 msec. The analytical data from Figure 18 are also included in Figure 21 for comparison. Figure 22 plots the mean response time to propagate an update from a group of \( m \) nodes to the parent node with the mean communication delay kept at 20 msec and the mean computation
Figure 21: Simulated Mean Response Time for Various Communication Delays
Comm Delay = 0.02 Sec

Mean response time (sec) x 10^{-3}

Figure 22: Simulated Mean Response Time for Various Computation Delays
Comm Delay = Comp Delay

Mean response time (sec) x 10^{-3}

Figure 23: Simulated Mean Response Time With Equal Computation and Communication Delays
delay varying from 100 msec to 500 msec. The analytical data from Figure 19 are also included in Figure 22 for comparison. Figure 23 plots the mean response time when the mean communication delay is equal to the mean computation delay and the delay varies from 5 to 40 msec. The analytical data from Figure 20 are also included in Figure 23 for comparison. The simulated curves are prefixed with SIM, and the analytical curves with Ana.

Examining the simulated data in Figure 21 reveals that the simulated data are very close to the analytical data and the simulated mean response time is slightly smaller than the analytical mean response time. The smaller simulated mean response time is reasonable, since in the analytical model a sibling of the triggering node advances a region only when the sync message from the triggering node for that region is received. In real operation, however, a sibling node may also advance its region due to its own local update that occurs concurrently or due to the receipt of a sync message for a later region from another node as demonstrated in Examples 4.3 and 4.3. When these are the cases, the sibling node will terminate the region earlier and thus shorten the response time.

Similar results can be observed in Figure 22 when the mean communication delay is kept at a constant and the mean computation delay is set at lower values. When the mean computation delays are high, however, the simulated data exceeds the analytical data for larger m. One possible explanation is the queuing of the updates. The time it takes to propagate one update upwards is around 530 (430) msec. When the mean update rate is 1 update per 100 sec with about 20 to 30 nodes, the probability that an
update occurs at a node before the node finishes processing the previous simultaneous region is increased. Since all processes are atomic, the update has to be queued until the previous simultaneous region is finished. Thus, the response time for the update is increased.

5.6 Simulation Data for Total Message Count

The overall performance of the distributed rule evaluation algorithm is affected not only by triggering updates but also by non-triggering updates, which do not toggle the global status of the rule query. For non-triggering updates, response time is not an issue since the global status is not changed. The messages generated by these non-triggering updates, however, add to the total number of messages that must be processed by the system. Depending on the urgency of the rule and the frequency the rule condition becomes true, the total message count may have a greater impact on the overall performance of the distributed rule evaluation algorithm than the response time for triggering updates.

The total message count is not easy to derive analytically since multiple updates may be combined into one at some internal node which is a direct ancestor for all nodes where updates occur. To derive the total message count, the update rate at a parent node is collected from the simulation, given the homogeneous update rate at \( m \) child nodes. For each update occurring to a node with \( m - 1 \) siblings, \( m - 1 \) sync messages and \( m \) data messages are required to propagate the update up one level. Let the update rate of this node be \( \mu/sec \), the number of messages generated by this node will be \( \mu(2m - 1)/sec \). Given a distributed rule tree, if the update rates of all
leaf nodes are known, the update rates of all internal nodes may be derived from the highest level up to the root node. The number of messages generated by all nodes can then be summed up to derive the total message count for the given distributed rule tree.

In Figure 24, the parent's update rates collected from the simulation are plotted with respect to the number of child nodes. Assuming all child nodes have the same update rate, the mean communication delay is kept at 20 msec and the mean computation delay at 100 msec. From the observation of Figure 24, the parent's update rate increases linearly as the number of child nodes increases. Closer examination reveals that the slope of the parent's update rate is approximately $\mu - 4 \ast p$, where $\mu$ is the child's update rate and $p$ is the probability that more than one update are captured in a simultaneous region.

The probability that more than one update are captured in a simultaneous region may be derived as follows. Given two sibling nodes A and B, assume an update occurs at Node A first. Let random variable $X$ represent the communication delay for the sync message sent from Node A to Node B for this update. Let random variable $Y$ represent the inter-arrival delay for an update to occur at Node B since the time the update occurred at Node A. $X$ has Exponential distribution with mean $\frac{1}{\lambda}$ and $Y$ has Exponential distribution with mean $\frac{1}{\mu}$. The probability that more than one update is captured in a simultaneous region between these two nodes is if $X \geq Y$. Assume
Simulated Update Rate at Parent Node

Update rate (/sec)

Figure 24: Parent's Update Rate
X and Y are independent. The joint pdf of X and Y can be derived. The probability that \( X \geq Y \) for any given x is

\[
f_{X \geq Y}(x) = \int_0^x f_{X,Y}(x,y) \, dy = e^{-\lambda x} - e^{-(\lambda + \mu)x}
\]

(5.19)

The probability that \( X \geq Y \) is

\[
p = \int_0^{\infty} f_{X \geq Y}(x) \, dx = \frac{\mu}{\lambda + \mu}
\]

(5.20)

With the parameter values we have been using, \( p \) is very small. Let \( \mu_i \) denote the update rate of a node with \( i \) child nodes which have a given homogeneous update rate \( \mu \). The update rate for a parent node with \( m \) child nodes, where \( m \geq 2 \), is

\[
\mu_m = \mu_2 + (m - 2) \cdot (\mu - 4 \cdot p).
\]

It can be observed from Figure 24 that \( \mu_2 < 2 \cdot \mu \). Thus, \( \mu_m \) is smaller than \( m \mu \).

5.7 Discussions of the Performance Data

The performance data of the distributed rule evaluation algorithm will be discussed from two aspects. First is to find better ways to organize nodes in a distributed rule tree. Second is to compare with rule evaluation algorithms in centralized active databases.

5.7.1 Node Arrangements in a Rule Tree

For a group of \( m \) nodes, there are many ways to arrange them into a tree. One extreme is to place all of them at one level with a common parent. The other extreme is to group every two of them at a time, introduce extra nodes as internal nodes, and build
Figure 25: Various Arrangements of $m$ Nodes
a binary tree of log $m$ levels. Figure 25 presents various re-arrangements of $m$ leaf nodes. Each arrangement is discussed below individually.

Case (a): For a one-level tree with $m$ leaf nodes displayed in Figure 25(a), $E_m$ denote the mean response time to propagate a triggering update from one of the leaf nodes to the root node, as derived in equation 5.13. The total number of messages required to propagate a triggering update are $m$ data messages and $m - 1$ sync messages.

Case (b): For the two-level tree with $m$ leaf nodes displayed in Figure 25(b), the mean response time to propagate a triggering update from one of the leaf nodes at the second level to the root node is $E_{m-1} + E_2$, and $m + 1$ data messages and $m - 1$ sync messages are required. Both the response time and the number of messages are larger than those of case (a) for any $m > 2$.

For the same tree to propagate a triggering update from the leaf node at level 1 up to the root node, the mean response time is $E_2$ and a total of 2 data messages and 1 sync message are all that are required. Both the response time and the number of messages are smaller than those of case (a).

The node singled out at level one is the only one that benefits from this arrangement of $m$ nodes.

Case (c): For the two-level tree with $m$ leaf nodes displayed in Figure 25(c), the mean response time to propagate a triggering update from one of the leaf nodes at the second level to the root node is $E_{m} + E_{m+1}$ and a total of $m + 1$ data messages and $m - 1$ sync messages are required. Both the response time and the number of messages are larger than those of case (a) for any $m > 2$. 
For the same tree to propagate a triggering update from one of the leaf nodes at level 1 up to the root node, the mean response time is $E_{m+1}$ and a total of $\frac{m}{2} + 1$ data messages and $\frac{m}{2}$ sync messages are required. Both the response time and the number of messages are smaller than those of case (a) for any $m > 2$.

The nodes that remain at level one are the ones that benefit from this arrangement of $m$ nodes.

Case (d): For the two-level tree with $m$ leaf nodes displayed in Figure 25(d), the mean response time to propagate a triggering update from one of the leaf nodes at the second level to the root node is $E_{\frac{m}{2}} + E_2$, and a total of $\frac{m}{2} + 2$ data messages and $\frac{m}{2}$ sync messages are required. Compared with case (a), the number of messages decreases while the mean response time increases for any $m > 2$.

Case (e): For a complete binary tree with $m$ leaf nodes as in Figure 25(e), the level of the leaf nodes is $\log m$. To propagate a triggering update from one of the leaf nodes to the root node, the mean response time is $\log m \times E_2$ and a total of $2 \times \log m$ data messages and $\log m$ sync messages are required. Compared with case (a), the number of messages decreases while the average response time increases for any $m > 2$.

From the above analysis, it can be concluded that response time increases as the height of the tree increases. The message count, however, is different. The update rates of the root node in any of the above case are all slightly smaller than $\frac{m}{2}$, which means that the total message count remains stable with the same number of leaf nodes. The message count for propagating triggering updates, however, increases or decreases depending on the arrangement of the tree. With a balanced multi-level tree,
the total number of messages required to propagate an update to the root node is less than that of a one-level tree with the same number of leaf nodes. With a skewed tree, more messages are required to propagate an update from a leaf node at a higher level than a one-level tree, but fewer messages are required to propagate an update from a leaf node at a lower level.

This observation suggests that for a group of $m$ nodes which have very different update frequencies, a skewed tree is probably a good choice. By arranging nodes with high update frequencies at a lower level and nodes with low update frequencies at a higher level, both the message count and response time from propagating triggering updates can be improved. For a group of $m$ nodes which have similar update frequencies, a fatter tree is a better choice than a taller tree. Since the total message count remains stable, a fatter tree has a shorter response time than a taller tree for triggering updates.

5.7.2 Comparison with Centralized Active Databases

When this distributed rule evaluation algorithm is compared with centralized rule evaluation algorithms, message count is not appropriate, because no message needs to be exchanged in a centralized environment. The only criteria for comparison is the response time.

Consider a rule decomposed into $m$ subrules, which are arranged into a one-level tree in this distributed evaluation algorithm. Assume local computation takes $P_i$ sec at node $i$, for any $1 \leq i \leq m$. When a triggering update occurs to node $i$, assume the response time for the root node to recognize the occurrence of this update is $E_m$,
then $E_m > P_i$ from the above performance study. The response time of this algorithm is compared with two types of centralized rule evaluation algorithms, the one which does not support rule decomposition [68] and the one which does [57].

Case: No rule decomposition is supported. For centralized active databases like POSTGRES [68], when one of the data referenced in a rule condition is updated, the entire rule query is evaluated. This strategy corresponds to performing all computations in all of $m$ subrules sequentially. Thus, the response time for this type of centralized rule evaluation will be $\sum_{i=1}^{m} P_i$. By examining Figure 19, it is noted that $E_m < m \times P_i$, for any $2 \leq m \leq 30$. Thus, the distributed rule evaluation algorithm is better than the centralized rule evaluation algorithms which do not decompose rules.

Case: Rule decomposition is supported. For centralized active databases that support rule decomposition like DIPS [57], intermediate results that partially match any rule condition are saved. When a triggering update comes in, only part of the rule condition needs to be recalculated. This strategy corresponds to evaluating only subrule $i$ which the triggering update affects. The response time for this strategy will be $P_i$, which is better than $E_m$ for the distributed evaluation algorithm.

Even though the distributed rule evaluation algorithm compares unfavorably with centralized rule evaluation algorithms which support rule decomposition, the difference, $E_m - P_i$, is the overhead incurred for the synchronization purpose as introduced by the distributed architecture. This overhead is about a few communication delays, even with larger $m$ as observed from Figure 18. If the rule is decomposed in such a way that the mean computation delay is significantly larger than the mean
communication delay, this overhead is a small price to pay to introduce production-style rules into the distributed databases for the benefits of local autonomy, incremental growth, reliability, and flexibility.
CHAPTER VI

A DISTRIBUTED RULE EVALUATION ALGORITHM CONSIDERING UPDATE RATES

6.1 Basic Idea

A distributed rule query is modeled as a relational expression tree, where leaf nodes are allocated on different sites and an internal node might be allocated on the same site as one of its children. In general, all nodes of a tree calculate some relational expressions and pass the results to their parents; specifically, a leaf node derives its relation from some local base relation, while an internal node derives its relation from relations received from its children. The relation derived by the root node represents the binding between the rule condition and the rule action. When this relation is not empty, the root node is responsible for alerting the system and forwarding this data to the rule action. The evaluation algorithm in a distributed environment determines when a rule query is true in a timely and consistent way with minimum communication overhead.

To minimize communication overhead, \( m \) child nodes of a one-level tree are arranged in a linear fashion to form a communication line. The nodes are arranged according to the rates updates occurring at these nodes. When a local update occurs,
a child node will recalculate its own status, taking into account the new update, and pass this new status to its parent node in an update message through this communication line. The child node with the lowest update rate will be the first in the communication line and will communicate its new status to its sibling with the next higher update rate. Whenever a node receives an update message from its sibling, it will pass the update message on to the next node in the communication line. The child node with the highest update rate will communicate directly with the parent node.

An update message received by an internal node from one of its children is considered a local update and the internal node will calculate the local expression to derive its new status. To propagate the update further up the tree, the internal node will form a linear communication line with its siblings at the same level. Eventually, the update will reach the root node. Figure 26 demonstrates how update messages are propagated in a multiple-level tree.

This evaluation algorithm is unfair to nodes with lower update rates because it will take a longer time for updates which occur at these nodes to propagate to the root node. However, since updates occur less frequently at these nodes, taking a longer time to react to updates that occur less frequently will be acceptable in many applications.

6.2 Procedure Pseudocodes

Assume a reliable underlying network which delivers messages in the order sent and loses no messages. Assume the following variables are maintained at each site and
initialized properly as follows. The variable site_id at each site is set to the id of the site. The variable next at each site is set to the site id of the sibling node which is next in the communication line. If there is no more sibling in the line, next is initialized to the negative site id of its parent node. For the root node, next is initialized to 0. The variable local.state is used to keep track of the status of the local expression. For internal nodes, an array child.state[i], where 1 ≤ i ≤ m and m is the number of its children, is used to keep track of the latest states received from child node i.

The procedure pseudocodes for the distributed rule evaluation algorithm are presented next. A leaf node will execute local.update() when a local update occurs and execute recv.sibling.update() when an update message is received from a sibling node. An internal node will execute recv.child.update() when an update message from a
child node is received and execute \textit{recv\_sibling\_update()} when an update message from a sibling node is received.

\textbf{Procedure local\_update()}

\/* Execute this procedure when a local update occurs at a leaf node */

Begin

\hspace{1em}\text{local\_state := eval(local expression);}

\hspace{1em}\text{send\_update(site\_id, local\_state);}  

End

\textbf{Procedure recv\_sibling\_update(originating\_site, recv\_state)}

\/* Execute this procedure when an update message */

\/* from a sibling node is received */

Begin

\hspace{1em}\text{send\_update(originating\_site, recv\_state);}  

End

\textbf{Procedure send\_update(originating\_site, current\_state)}

\/* This procedure is called from local\_update() */

\/* recv\_sibling\_update() or recv\_child\_update() */

Begin

\hspace{1em}\text{if next = 0 then}
/* this is the root node */

if current.state.status = True then
    Begin
        recognize the occurrence of the rule query;
        pass current.state to rule action;
    End

else if next < 0 then
    /* no more siblings to pass on status */
    send_parent_update(-next, current.state);

else

    /* pass on the status to the next sibling node */
    send_sibling_update(next, originating_site, current.state);

End

Procedure recv_child_update(originating_site, recv_state)
/* Execute this procedure when an update message */
/* from a child node is received */

Begin
    child_state[originating_site] := recv.state;
    local.state := eval(child_state);
    send_update(site_id, local.state);

End
6.3 Arguments for Correctness

Assume that the underlying communication network is reliable, loses no messages, and delivers messages in the order sent. Assume that all procedures are executed as atomic processes. The distributed rule evaluation algorithm presented above has the following properties.

Property 1: Every update will arrive at the root node. Since each update occurred at a leaf node will propagate to its parent node through a sequence of update messages and the underlying network is assumed to lose no messages, each update will arrive at its parent node. Since each update arrived at an internal node will trigger an evaluation at the node and the new state will be propagated to its parent node in a sequence of update messages, this update will arrive at the parent of this internal node. Eventually, every update will arrive at the root node.

Property 2: All updates are ordered by their arrival time at the root node. Since all procedures are atomic, the arrival time of each update at the root node is unique. Every update will arrive at the root node from Property 1 and a total order exists among all possible times at the root node. Thus, all updates are ordered by its arrival time at the root node.

Property 3: For each update arrived at an internal node, a consistent set of child states is used to derive the combined state. Whenever an update arrives at an internal node, the new state for the child node where the update occurred will be recorded in the child_state array. The local expression at the internal node is re-evaluated with the states stored in child_state. Since child_state keeps track of a
consistent set of states for its child nodes with the latest states as received from its child nodes, a consistent set of child states is used to derive the combined state for each update arrived at an internal node.

Based on these properties, we can show that the distributed rule algorithm is correct, that is, it exhibits no erroneous behavior. A distributed rule evaluation algorithm behaves erroneously if either of two conditions can arise:

• a rule condition is true but is not recognized; and

• a rule condition is false but is recognized as true.

Claim 1: Every true status of a rule query will be recognized. Denote the update that changes the status of a rule query to true the triggering update. Since every update will arrive at the root node from Property 2, this triggering update will reach the root node. Since a consistent set of child node states will be used to derive the combined state at an internal node for any update arrived at an internal node from Property 3, the global state at the root node will be derived from a consistent set of states of all nodes in the tree for the triggering update. The true status triggered by the update will thus be recognized when the triggering update reaches the root node.

Claim 2: No false status of a rule query will be recognized as true. From Property 3, a global state as affected by an update is derived from a consistent set of states of all nodes in the tree when the update arrives at the root node. If the global state corresponding to an update is evaluated to be false, the root node will recognize it and will not report true to the system.
CHAPTER VII

PERFORMANCE STUDY OF THE DISTRIBUTED RULE EVALUATION ALGORITHM CONSIDERING UPDATE RATES

For databases whose rules are not changed very often, the performance of the distributed rule evaluation algorithm plays the key role in the overall performance of the distributed active databases, since the rule decomposition and distribution algorithms are executed at rule definition time, while the distributed rule evaluation algorithm is executed constantly as long as the database is operational. The performance of this algorithm is analyzed next in terms of the message count and the response time.

7.1 Notations and Definitions

To study the performance of the distributed rule evaluation algorithm, a distributed rule condition is modeled as a tree of nodes. Database updates may occur to any leaf nodes. The leaf nodes where the update occurs is termed the *triggering leaf node*. The direct ancestors of the triggering leaf node will become *triggering nodes* when they receive the update messages corresponding to the update.
Definitions: Among a group of $m$ sibling nodes, number the node with the $i$th highest update rate $i$. Node 1, thus, has the highest update rate while Node $m$ has the lowest. For simplicity, let Node 0 denote the parent of these $m$ nodes. Let $\mu_i$, $1 \leq i \leq m$, denote the update rate at Node $i$, and $\mu_p$ denote the update rate at the parent node. Node $i$ is an older sibling of Node $j$ if $i < j$.

For an update to propagate to the root node, the triggering leaf node, its direct ancestors, and the older siblings of these nodes are all involved in the evaluation process. We assume that updates arrive at a node according to Poisson distribution [17]. The message count and the response time to propagate an update from a group of $m$ sibling nodes to their parent node and the update rate at the parent node given child nodes’ update rates are analyzed below.

7.2 Derivation of Message Count

For node $i$ to propagate an update to the parent node, the update will travel from node $i$ to the parent node through nodes $i-1, i-2, \ldots, 1$. Therefore, $i$ update messages are required. The mean number of messages to propagate an update from a group of $m$ sibling nodes, each with update rate $\mu_i$ is

$$\frac{\sum_{i=1}^{m} i \mu_i}{\sum_{i=1}^{m} \mu_i} \quad (7.1)$$

If the update rates at all sibling nodes are the same, $\mu_i = \mu \forall i$, this mean message count is

$$\frac{\mu \sum_{i=1}^{m} i}{m \mu} = \frac{m + 1}{2} \quad (7.2)$$
7.3 Derivation of Parent’s Update Rate

Since each update is transmitted without alteration by the older sibling nodes of the originating node, every update occurring at one of the child nodes will be perceived as a separate update to the parent node. Therefore, the update rate at the parent node is

$$\mu_p = \sum_{i=1}^{m} \mu_i$$  \hspace{1cm} (7.3)

When the update rates at all sibling nodes are the same, \(\mu_i = \mu \forall i\), the parent’s update rate will be \(m\mu\).

Given a tree of nodes, if the update rates of all leaf nodes are known, the update rates of internal nodes can be derived according to equation 7.3 from the highest level down to the root node. The number of messages generated by each node can be derived according to its position in the communication line with its siblings according to equation 7.1. The total number of messages generated by this algorithm to process all updates is the summation of the products of the update rate and the message count for all nodes of the tree.

7.4 Derivation of Response Time

The response time to propagate an update up a level will be measured as the time elapsed from the moment the update arrives at the node to the moment the update arrives at the parent of the node.

Let random variable \(Z_i\) denote the response time for node \(i\) to propagate an update to the parent node. Let random variable \(Y\) denote the local computation delay to
process this update at node \(i\). Let random variables \(X_j, 1 \leq j \leq m\), denote the communication delay to pass this update from node \(j\) to node \(j - 1\). Assume all \(X_j\)'s have Exponential distributions with mean \(\frac{1}{\lambda}\). Assume \(Y\) has an Exponential distribution with mean \(\frac{1}{\lambda'}\). Then \(Z_i = Y + \sum_{j=1}^{i} X_j\).

Since \(X_j\)'s are independent and have Exponential distributions with mean \(\frac{1}{\lambda}\), the sum \(\sum_{j=1}^{i} X_j\) has a Gamma distribution with parameters \(i\) and \(\lambda\) [13]. The mean of the sum \(\sum_{j=1}^{i} X_j\) is \(\frac{i}{\lambda}\) [17]. Since \(Y\) is independent with \(\sum_{j=1}^{i} X_j\), the mean of \(Z_i\) is thus \(\frac{1}{\lambda'} + \frac{i}{\lambda}\).

The mean response time for the parent of \(m\) nodes is

\[
ET = \frac{\sum_{i=1}^{m} \mu_i \left(\frac{1}{\lambda'} + \frac{i}{\lambda}\right)}{\sum_{i=1}^{m} \mu_i}
\]

\[
= \frac{1}{\lambda'} + \frac{1}{\lambda} \sum_{i=1}^{m} \frac{i \mu_i}{\lambda \sum_{i=1}^{m} \mu_i}
\]

(7.4)

When \(\mu_i = \mu \ \forall i\), the mean response time to propagate an update to the parent of \(m\) nodes is

\[
ET = \frac{1}{\lambda'} + \frac{1}{\lambda} \frac{m \mu (m+1)}{2}
\]

\[
= \frac{1}{\lambda'} + \frac{m+1}{2\lambda}
\]

(7.5)

7.5 Discussions of the Performance Data

In the following subsection, the performance data of the distributed rule evaluation algorithm from the above analysis are compared with simulation data of the current approach and with the distributed rule evaluation algorithm using the simultaneous region approach presented in Chapter 4.
7.5.1 Comparison Between the Analytical Model and Simulation

The distributed rule evaluation algorithm is also simulated using SIMPAS, an event-driven discrete system simulation language based on PASCAL. The simulated parent's update rates and mean response time are compared with the analytical data in Figures 27 and 28 with the analytical curves prefixed with "Ana" and simulated curves with "SIM". To derive concrete numbers for easy comparison, the mean update rates among $m$ nodes are assumed to increase with a factor of 0.9 and the maximum mean child update rates are varied from 0.01/sec to 0.1/sec in Figure 27 and is kept at 0.01/sec in Figure 28. In Figure 28, the mean computation delay is kept at 0.1/sec while the mean communication delay is varied from 0.01 to 0.05/sec. Examining Figures 27 and 28 reveals that the simulated data are very close, almost identical, to the analytical data, which confirms the correctness of our analytical model.

7.5.2 Comparison with the Distributed Rule Evaluation Algorithm Using Simultaneous Regions

This distributed rule evaluation algorithm is also compared with the simultaneous region approach discussed previously in Chapter 4, which arranges $m$ subrules as a one-level tree with $m$ child nodes. The parent's update rates from both algorithms are compared in Figure 29, while the mean response times to propagate an update one level up are compared in Figure 30. The curves for the simultaneous region approach are suffixed with "SR". The update rates among child nodes are assumed to increase by a factor ranging from 0.5 to 1 and the maximum mean child update
Parent’s Update Rate (Factor=0.9)

Figure 27: Comparison in Parent’s Update Rate between Analysis and Simulation
Figure 28: Comparison in Response Time between Analysis and Simulation
rate is set at 0.01/sec in both Figure 29 and Figure 30. When the increasing factor
is 1, the update rates are actually identical among child nodes. Moreover, the mean
computation delay is kept at 0.1sec and the mean communication delay is kept at
0.02sec.

Both approaches have about the same parent's update rate when the update
rates among child nodes increase with a factor less than 1. When the update rates
among child nodes are uniform, current approach has slightly higher update rates at
the parent node than the simultaneous region approach. The difference in parent's
update rate, however, ranges from barely noticeable to about 7% as the number of
child nodes approaches 30. For each update arrived at a parent node with $m$ child
nodes, at least $m$ data messages and $m - 1$ sync messages are exchanged for the
simultaneous region approach. In contrast, only 1 to $m$ messages are exchanged with
a mean at $\frac{m+1}{2}$ for each update arrived at the parent node for the current approach.
Since the total message count is the product of the update rate and the message
count, the total message count for the current approach is less than the total message
count for the simultaneous region approach by an order less than 4.

In examining the response time, the current approach in general has longer mean
response time than the simultaneous region approach. Note that the mean response
time to propagate an update up one level for the simultaneous region approach is
almost the same regardless how the update rates vary among the child nodes; whereas,
the mean response time for the current approach sharply decreases as the increasing
factor of child update rates decreases. In other words, how fast the mean response time
Comparison of Parent’s Update Rate

Figure 29: Comparison in Parent's Update Rate With Simultaneous Region Approach
Figure 30: Comparison in Response Time With Simultaneous Region Approach
to propagate an update up one level decreases depends on how fast the update rates among child nodes decrease for the current approach. The mean response time for the current approach is at most three times the mean response time for the simultaneous region approach even in the worst case, when the child update rates are identical. When the child update rates increase moderately at 0.9, the mean response time for the current approach is at most twice that for the simultaneous region approach. As the child update rates decrease more drastically, the difference in the mean response times between the two approaches decreases further.

In conclusion, the simultaneous region approach is insensitive about the update rate variances among child nodes while the current approach takes advantage of the variances. Moreover, the current approach exchanges fewer messages by a factor less than 4 but has a longer mean response time by a factor less than 2 than the simultaneous region approach. The tradeoff between the number of exchanged messages and the mean response time can be determined by the frequency of how often a rule becomes true and the requirement of how soon a rule should be recognized once it becomes true.
CHAPTER VIII

CONCLUSIONS

With the integration of production-style rules, active databases have recently emerged as one of the most important frontiers for the next-generation DBMS's [43,55,59,66]. In addition to more and larger applications which cannot be supported with the current production systems, active databases gain a unifying mechanism for traditional database functionalities which are implemented in an ad hoc manner. Numerous applications for active databases have been identified and a lot more remain to be discovered.

Rule monitoring, however, places a tremendous performance requirement on the success of the active databases. Multiple sites available in a distributed database offer an opportunity to parallelize this rule monitoring process. On the other hand, distributed databases are gaining popularity due to the genuine needs they fulfill for enterprises which are geographically distributed. The active features offered by the production-styles rules are desirable for distributed databases.

The problems in distributed active databases have just started to receive attention. This dissertation is one of the pioneers in addressing problems in distributed active databases. In the following sections, major contributions of this dissertation are summarized and future research directions are discussed.
8.1 Summary

Three design issues for monitoring rules in a distributed active database were identified in this dissertation. The first issue was how to decompose a rule, and the second issue was how to distribute a decomposed rule and base relations to sites so that rule monitoring could be carried out in multiple sites in parallel. The third issue had to do with collecting local results from multiple sites and deriving a global status from these local results. The first two issues have been addressed in different contexts, namely parallel production systems and parallel deductive databases. The third issue, however, is unique to the distributed environment due to the geographical separation of sites and the lack of a global clock.

An integrated approach was proposed to address these three issues together in this dissertation. A decomposition algorithm was proposed to address the rule decomposition issue, and a distribution algorithm was proposed to address the rule distribution issue. Finally, two distributed evaluation algorithms were proposed to monitor decomposed complicated rules in a distributed active database. The performance study of the two distributed evaluation algorithms provide feedback for the first two algorithms.

The decomposition scheme represented a rule query in a relational operator tree and used algebraic manipulations of the relational operator tree to optimize the decomposition. A new relational operator, AND, was proposed to identify independent parts of a rule query in order to reduce the amount of data to be transferred between sites. Algebraic manipulations of the AND operator with the rest of the relational
operators were presented. Multiple relational operators were grouped into a subrule.
Thus, the decomposition level of this algorithm falls between the relation level and the relational operator level.

In a distributed database, base relation distribution is less adaptable than rule distribution due to the many complicated issues in distributing the base relations. The proposed distribution algorithm did not require the reorganization of the base relations. Each base relation was assumed to be partitioned horizontally, vertically, or a combination of both. A subrule from the above decomposition algorithm would be distributed to the sites the referenced base relation allocates in compliance with the partitions.

The distributed evaluation algorithms aimed to derive a correct and consistent global rule evaluation result, which is difficult to achieve due to the distributed environment. Two distributed evaluation algorithms were proposed. The first algorithm used the simultaneous region approach, which would combine the effects of multiple concurrent updates. The second algorithm ordered every update according to time of occurrence and update frequencies.

The performance of both algorithms was studied in terms of the number of messages and the mean response time to propagate an update up to the root node and the total message count in operating the algorithms. Both analytical models and simulations were used in the performance study.

In terms of response time, the first algorithm compares favorably with centralized rule monitoring algorithms which do not decompose rules. For centralized rule
monitoring algorithms which use decomposition, our algorithm takes longer time. The difference in time is incurred for the synchronization purpose, which is necessary in a distributed environment. This overhead is about a few communication delays which is small if the mean computation time at a node is significantly larger than the mean communication delay of a single message.

The performance data of the first algorithm suggested better ways to arrange a fixed number of subrules. When the update frequencies of the subrules are not very different from each other, a fatter tree is better than a taller tree. In contrast, when the update frequencies of the subrules are very different, a skewed tree arranged by placing subrules with high update frequencies near the root and placing subrules with low update frequencies near the leaf of the tree is a good choice.

When a rule does not become true very often, the exchanged messages in order to propagate every update to the root node become an extra burden for the system to process without gained productivity. To reduce the number of exchanged messages, the second distributed rule evaluation algorithm is proposed. The total message count used by this algorithm is less than that of the first algorithm by a factor slightly less than 4. However, the mean response time of the second algorithm is larger than that of the first algorithm with a factor ranging from 3 to 1.2 depending on how fast the update rates decrease among the child nodes. When the update rates among the child nodes decrease drastically, with an increasing factor less than 0.5, the mean response time of the second algorithm is smaller than that of the first algorithm when the number of child nodes is large.
8.2 Future Work

This dissertation is the first step in addressing problems in distributed active databases. A lot more problems remain open and need to be investigated before distributed active databases become a mature technology. The following discussions point out some possible future research directions.

The assumption used in the rule distribution algorithm is that each base relation is partitioned into fragments and distributed without duplication. The cases with fully or partially replicated base relations are not investigated. Due to the wide range of possible usages of rules, the rule monitoring in such an environment is an interesting problem. If the rules are used in the conventional way, the problem becomes how to derive a rule query status from multiple copies of potentially inconsistent data. How to reach the consensus under such a situation is worth further investigation. Moreover, rules can also be used to enforce consistency among multiple copies of the same data. The syntactical problem of how to use rules to enforce multiple copy consistency, however, is not clear at this point. The mechanisms to address each copy and to decide which value is prevalent need to be solved before the rule monitoring problem can be attacked in this case. The problem can be further extended to heterogeneous databases, where rules are used to enforce semantic consistency, which also remains open at this point.

This dissertation investigated intra-rule optimization strategy for complicated rules in a distributed environment. When multiple rules are considered together, inter-rule optimization strategy needs to be looked into. One strategy is to borrow
techniques from multiple query optimization [54]. Since rules are less likely to change than user queries, finding common subexpressions among rules will be easier than finding common subexpressions among an arbitrary number of user queries. The Rete and the TREAT algorithms, used in production systems, offer another alternative for inter-rule optimization. However, due to the limited expressive power offered in production systems, these algorithms need to be extended to fully support rules in a DBMS environment.

Another extension when considering inter-rule optimization is how to fit the rule monitoring into the transaction model. The work reported in [6] is the attempt to integrate the entire rule processing cycle into the transaction model. Rule monitoring is just one phase of the entire rule processing cycle, which consists of rule monitoring, conflict resolution, and rule action execution. As the time spent in the monitoring phase is reduced, the other phases need to be investigated for overall performance improvement. Nevertheless, the assumption used in [6] is that a base relation is allocated at a single site. Partitioned or replicated relations are not considered. How to distribute the entire rule processing cycle but remain at a fine granularity to achieve better performance remains open.
Appendix A

ALGEBRAIC LAWS FOR RELATIONAL OPERATORS

Algebraic laws in manipulating regular relational operators as introduced in [70] are presented in this appendix to complete the presentation of the decomposition algorithm. Conventional notations are used for these relational operators, such as selection($\sigma$), projection($\pi$), Cartesian product($\times$), join($\bowtie$), union($\cup$), intersection($\cap$), and set difference(−). Let $E$ or $E_i$, for any $i$, stand for some relational expression; $F$ or $F_i$ for some condition; $A_i$, $B_i$, and $C_i$, for some attribute names.

Laws involving joins and Cartesian Products

1. Commutative laws for joins and products.

$$E_1 \bowtie_F E_2 \equiv E_2 \bowtie_F E_1$$

(A.1)

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

(A.2)

$$E_1 \times E_2 \equiv E_2 \times E_1$$

(A.3)

2. Associative laws for joins and products.

$$(E_1 \bowtie_{F_1} E_2) \bowtie_{F_2} E_3 \equiv E_1 \bowtie_{F_1} (E_2 \bowtie_{F_2} E_3)$$

(A.4)
Laws involving selections and projections

3. Cascade of selections and projections.

\[ \sigma_{F_1}(\sigma_{F_2}(E)) = \sigma_{F_1 \land F_2}(E) \] \hspace{1cm} (A.7)

\[ \sigma_{F_1}(\sigma_{F_2}(E)) = \sigma_{F_2}(\sigma_{F_1}(E)) \] \hspace{1cm} (A.8)

If \( A_1, \ldots, A_n \) is a subset of attribute names among \( B_1, \ldots, B_m \),

\[ \pi_{A_1, \ldots, A_n}(\pi_{B_1, \ldots, B_m}(E)) = \pi_{A_1, \ldots, A_n}(E) \] \hspace{1cm} (A.9)

4. Commuting selections and projections.

If condition \( F \) involves attributes only from \( A_1, \ldots, A_n \),

\[ \pi_{A_1, \ldots, A_n}(\sigma_F(E)) = \sigma_F(\pi_{A_1, \ldots, A_n}(E)) \] \hspace{1cm} (A.10)

If condition \( F \) involves attributes \( B_1, \ldots, B_m \) that are not among \( A_1, \ldots, A_n \),

\[ \pi_{A_1, \ldots, A_n}(\sigma_F(E)) = \pi_{A_1, \ldots, A_n}(\sigma_F(\pi_{A_1, \ldots, A_n, B_1, \ldots, B_m}(E))) \] \hspace{1cm} (A.11)

5. Commuting selections and projections with a Cartesian product.

If all the attributes mentioned in \( F \) are attributes of \( E_1 \),

\[ \sigma_F(E_1 \times E_2) = \sigma_F(E_1) \times E_2 \] \hspace{1cm} (A.12)

If \( F = F_1 \land F_2 \), and \( F_1 \) only involves attributes of \( E_1 \), and \( F_2 \) involves only attributes of \( E_2 \),
\[ \sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2) \quad (A.13) \]

If \( F = F_1 \land F_2 \), and \( F_1 \) only involves attributes of \( E_1 \), but \( F_2 \) involves attributes of both \( E_1 \) and \( E_2 \),

\[ \sigma_F(E_1 \times E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1) \times E_2) \quad (A.14) \]

If \( A_1, \ldots, A_n \) is a list of attributes of which \( B_1, \ldots, B_m \) are attributes of \( E_1 \), and the remaining attributes, \( C_1, \ldots, C_k \), are from \( E_2 \),

\[ \pi_{A_1, \ldots, A_n}(E_1 \times E_2) \equiv \pi_{B_1, \ldots, B_m}(E_1) \times \pi_{C_1, \ldots, C_k}(E_2) \quad (A.15) \]

6. Commuting selections and projections with a union.

If we assume that the attributes of \( E_1 \) and \( E_2 \) have the same names as those of \( E \), or at least there is a given correspondence that associates each attribute of \( E \) with a unique attribute of \( E_1 \) and a unique attribute of \( E_2 \). Then we may write

\[ \sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2) \quad (A.16) \]

\[ \pi_{A_1, \ldots, A_n}(E_1 \cup E_2) \equiv \pi_{A_1, \ldots, A_n}(E_1) \cup \pi_{A_1, \ldots, A_n}(E_2) \quad (A.17) \]

7. Commuting selections with a set difference.

With the same assumption as above,

\[ \sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2) \quad (A.18) \]


