EXTENDED RELATIONAL ALGEBRA (ERA) FOR SITUATION OPTIMIZATION IN ACTIVE DATABASES

By

SUDHANSHU GARG

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1992

Dedicated to the fond memories of my uncle,
Surendra K. Garg

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser, Dr. Sharma Chakravarthy, for showing me the path of research which was just a fantasy before, and for giving me an opportunity to work on the challenging *Sentinel* project. I am extremely grateful to Dr. Stanley Su and Dr. Ravi Vardarajan for serving on my committee and for their comments to improve the manuscript.

Many thanks are due to Mrs. Sharon Grant. I remember a sentence that might give insight into her indefatigable efforts in providing a well administered research environment, “Please use only one filter at a time,” a note above the coffee pot in the Database Systems R&D Center. I am grateful to Mrs. Marlene Hughes for all the help and cooperation at the time of need.

I will also take this opportunity to thank all the graduate students at this center for their help and friendship. I am proud of their efforts in establishing this as a preeminent research group of the nation in databases, and also in making it a rendezvous of diverse cultures and experiences. I thank Louiqa, Ping Wu, Tarik Anwar and Arun Gopalan for many fruitful technical discussions, and Sid, Ashish, Roy, Ravi, Jay, Whang, Deepak, Shashi, Shyam, Soon, Charles and many more for comic relief.

Last, but not the least, I thank my parents and sisters for their love. Without their encouragement and endurance, this work would not have been possible. Creativity, critical understanding, logical analysis and dedication to purpose — I learned from my father — played an indispensable background role in making this work happen.

I am especially grateful to Dr. Brij Mohan, Dr. J.D. Sharma, Sanjay Gosain and Kishore for teaching me a lot of things about myself and life.

This work is supported by the NSF grant IRI-9011216 and the Florida High Technology and Industrial Council grant UPN# 90091013. I am grateful to both and specifically to the loving and caring people of Florida for the latter.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vii

ABSTRACT viii

CHAPTERS 1

1 INTRODUCTION 1

2 SURVEY OF RELATED WORK 5

2.1 Expressiveness 7
2.2 Execution Semantics 9
2.3 Management of Rules 12
2.4 Situation Optimization 12
2.4.1 Optimization in ETM 13
2.4.2 Optimization Techniques in HiPAC 14
2.4.3 POSTGRES (PRSII) Optimization Techniques 16
2.4.4 Ariel Rule Optimization 18
2.4.5 Optimization in AI Systems 19

3 PROBLEM STATEMENT AND APPROACH 20

3.1 Motivation 20
3.2 Requirements 20
3.3 Approach 21

4 SENTINEL EXTENDED RELATIONAL ALGEBRA 22

4.1 Basic Definitions 22
4.2 Operators of ERA 25
4.2.1 IncrSelect 25
4.2.2 IncrProject 27
4.2.3 IncrJoin 31
4.2.4 IncrUnion 34
4.2.5 IncrDiff 36

5 COMPUTING CHANGES TO COMPLEX EXPRESSIONS 42

5.1 Optimization 42
5.2 Generic Chain Rule 43
5.3 Transformations of Incremental Operators 45
5.4 Proofs of Incremental Transformations 46
5.5 Example 49

6 CONDITION EVALUATION IN DISTRIBUTED ENVIRONMENT . . . 54

6.1 Distributed Active Database Issues 55
6.2 Horizontal Fragmentation 58
6.3 Vertical Fragmentation 58
6.4 Hybrid Fragmentation 59

7 CONCLUSIONS AND FUTURE WORK 60

APPENDIX 63

REFERENCES 68

BIOGRAPHICAL SKETCH 73

LIST OF FIGURES

2.1 Rule Format Possibilities. 8

4.1 Structure of delta. (a) A transaction, T, on EMP. (b) A relation, EMP. (c) EMP after the transaction. (d) Delta relation of EMP. 39

4.2 Algorithm for Evaluating Changes to a Project View. 40

4.3 Shows overlap of deltas and relations for IncrJoin operator. 40

4.4 Shows overlap of deltas and relations for IncrUnion and IncrDiff operators. 41

5.1 Two approaches for optimization. 51

5.2 Illustration of the Generic Chain Rule. 52

5.3 Illustration of the two Optimization Approaches. 53

A.1 Structure of HiPAC's delta. (a) A transaction, T, on EMP. (b) A relation, EMP. (c) EMP after the transaction. (d) Delta relation of EMP. 67

Abstract of Thesis
Presented to the Graduate School of the University of Florida
in Partial Fulfillment of the Requirements for the
Degree of Master of Science

EXTENDED RELATIONAL ALGEBRA (ERA) FOR SITUATION OPTIMIZATION IN ACTIVE DATABASES

By

Sudhanshu Garg

May, 1992

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Sciences

An active DBMS accepts rules composed of *situations* (event and condition components) and *actions*, and monitors situations to execute the corresponding actions. In this thesis, we address the problem of situation monitoring and propose solutions for optimizing conditions, in centralized as well as distributed databases.

Situation-action rules often monitor changes to tuples, stored relations and also derived relations, as well. Efficient evaluation of situation-action rules requires a general framework for computing changes and performing optimizing transformations on the expressions involving changes. This thesis discusses an algebraic approach for accomplishing this.

For optimizing conditions in the presence of changes, we have proposed a generalized chain rule that works on n-ary operators. Incremental versions of all the relational operators have been developed. Optimizing transformations over the incremental operators have also been developed. For the evaluation of changes in the

distributed case, we have incremental form of the Union operator. Incremental form of Set Difference has also been added, and the incremental form of IncrJoin has been redefined, removing the problems in the earlier definition. A general version of Incremental Project has been developed.

CHAPTER 1
INTRODUCTION

Traditional DBMSs are *passive*, in the sense that they execute queries or transactions only when explicitly requested to do so by a user or an application program. Numerous applications, such as, process control, factory automation, office work-flow control, computer aided design (CAD), computer integrated manufacturing (CIM), patient monitoring, network management, power generation/distribution, air-traffic control, battle management, stock exchange etc. require automatic and timely response to critical and non-critical situations which is not well supported by these passive DBMSs. These applications require automatic monitoring of situations defined over the state of the database and a capability of taking actions when the state of the underlying database changes. For example, in a data/voice network, we need to monitor various types of situations such as link overloading, hardware failures, network partitioning, unauthorized accesses (e.g. line tapping, accessing privileged services), delinquency of accounts, connection duration, etc. Whenever such situations are detected, an action must be fired automatically, such as in the case of hardware failure, depending upon the availability, activate the standby hardware modules; choosing alternate paths in the case of link overloading or breakdown; sending a voice/data mail to delinquent account holders; disconnecting the subscribers with long overdue payments; performing modifications in the billing information when a bill is paid and subtracting the discounts available to the subscriber, modifying the account parameters when class of service of a subscriber changes, etc.

Design databases (CAD) are characterized by non-trivial requirements for consistency enforcement which arise from complex structure of data objects, intricate

dependencies among objects, and long transactions. These databases require that the global consistency be reached stepwise through different stages of local consistencies. Further, time and extent of consistency checking and how to react to consistency violations needs to be flexible and determined dynamically, in some cases under user control and in others under system control. Transaction as a unit of consistency enforcement is insufficient in these cases. Traditional approach of integrating consistency control inside the transaction manager is very restrictive and needs to be enhanced with the help of a general purpose mechanism [40, 26].

There exist two approaches to enhancing the capabilities of a passive DBMS to meet the requirements of non-traditional applications, however neither is satisfactory. The first approach is to write an application program that periodically queries (polls) the database to determine if the situation being monitored has occurred. The problems with this approach are, first, if polling is too slow, there is a risk of missing the response window, and second, if polling be too frequent, there is a risk of flooding the system with queries that usually return an empty answer, thus deteriorating the system performance. The second approach is to augment each program that updates the database to check the situation being monitored and take appropriate action if the situation turns out to be true. The problem with this approach is that it compromises software modularity, code reusability and optimizability. This approach is prone to duplication of code and effort in different application programs and on part of different users. Software modularity is compromised as any modification to the situations being monitored or to the corresponding actions will require modifying every application program that updates the database. Another disadvantage of this approach is that finding an optimized way of evaluating situation is burdened upon the application programmer.

An active DBMS provides both modularity and timely response. It, in addition, shifts the burden of optimizing situations from the programmer to the system. It provides full functionality of a traditional DBMS and is capable of reacting automatically and asynchronously to situations occurring in its environment due to database state changes. For this purpose, situation monitoring needs to be incorporated as an integral part of the DBMS. Active DBMSs accomplish this by adding **E**vent-**C**ondition-**A**ction rules (ECA rules) to specify events to be monitored, condition to be evaluated when the appropriate events occur, and action to be invoked if the condition evaluates to true. They allow events and conditions (jointly called *situations*) and actions to be specified declaratively to the system. The system now bears the responsibility of efficient detection of events, evaluation of conditions and execution of the corresponding actions, all automatically and without requiring user intervention.

Rule evaluation can be divided into three basic activities: event detection, condition evaluation and execution of action. An *event* can either be a database event (insert/delete/update) or a non-database event (e.g., external or temporal) which is an instantaneous occurrence on the time line [17]. A *condition* can be considered as a side-effect free query defined over state(s)[1] of the database which decides whether the rule is applicable or not. If the query returns a non-empty result, it means that the rule can be fired. Otherwise, the rule cannot be fired. An *action* is like a transaction which performs one or more operations on the database.

Efficiency of rule evaluation depends primarily on optimization of the condition part and that of event detection algorithms. Efficient event detection algorithms are proposed by Chakravarthy and Mishra [17]. The action part of a rule is a transaction (e.g. containing TRANSACT-SQL commands) which gives fewer opportunities for optimizations, similar to what used for transactions [52], and is not side-effect free

[1] For example, an update transaction brings the database state from one state, say S1, to another state (say, S2). This makes both pre- and post- states of the database relevant to conditions.

unlike situations. In this thesis, our emphasis is on optimizing the condition part in centralized as well as distributed environments.

The rest of this thesis is organized as follows: in Chapter 2, we present a brief survey of active databases. We analyze different research and commercial efforts in the field of active databases, with special focus on their optimization techniques. Chapter 3 contains motivation for this work and our approach to condition evaluation and optimization. In Chapter 4, we present an algebra for evaluating changes to stored/derived relations. It includes incremental forms of Select, Project, Join, Union and Set Difference. It is shown that these operators overcome the problems present in earlier work. Proofs of these operators are given. In Chapter 4, we present two ways for optimizing condition expressions. Equivalence transformations over the Incremental Operators are also presented. A Generic Chain Rule, that extends the earlier Chain Rule to allow n-ary operators, is introduced for evaluating changes to complex relational expressions. Chapter 5 discusses condition-optimization in the presence of distributed relations. Various issues involved in this are high-lighted and the cases of horizontal, vertical and hybrid fragmentation are addressed. Chapter 6 contains our concluding remarks and suggestions for future research. A comparison with the HiPAC approach is made in Appendix.

CHAPTER 2
SURVEY OF RELATED WORK

Appearance of primitive *active* capabilities can be tracked back to as early as in the first database management systems, such as, ON clause of CODASYL [47] and triggers in System R [28, 27]. However, the active functionality of these systems was limited to enforcement of integrity constraints. The term *active* was first used by Morgenstern [44] to describe a system that supports automatic update of views and derived data as base data are updated. Use of triggers for materialized view maintenance has been of interest to many researchers [7, 39, 50, 44, 4, 37, 41, 33, 9]. Triggers that fire on a specified time, called time-triggers, have been used in office system applications [66, 3].

Rules have been supported in another branch of databases, deductive database systems, also. However, deductive databases are passive as they only respond to explicit queries from the outside world. The deductive rules supported in these systems are based on LUSH clause subset of first order predicate logic, such as in PROLOG. These rules are capable of specifying properties about the data and are equipped with a backward-chaining deductive inference engine for processing recursive queries [2, 12, 20, 46]. It has been suggested that deductive and active database rule systems compliment each other [36].

Production rules (e.g., rules in OPS5 [31, 30]), daemons (e.g., in ARS [65]), active objects [6] and procedural attachments to slots of frames [42, 5] have been used in AI systems as active knowledge representation and inferencing mechanisms. Expert systems based on these mechanisms are inherently main memory based and capable of handling only a small number of rules and facts. Another limitation of these systems

is that they typically allow only single threaded execution of rules. This limits their importance to active database systems which require handling a large database and a rule-base, stored on secondary memory. Many researchers have proposed effective extensions to AI production systems to relax the main memory restriction and to support large number of rules and facts [53, 54, 43, 48]. There has been some work on translating OPS5 rules into active database rules [15], also.

Recent work on active database management systems is geared toward integrating rules/triggers/alerts with the DBMS at the system level. Some of the prominent research prototypes are HiPAC (Xerox Advanced Information Technology [11, 13, 23]), POSTGRES (University of Berkeley [57, 59, 60, 58]), Starburst (IBM Almaden Research Center [64, 63]), Ariel (Wright State University [35, 34]) and ETM (Karlsruhe University [26, 40]. There is also some work on incorporating rules in semantic and object-oriented data models [22, 62, 32, 48, 55, 19].

In the last few years, many of the commercially available relational products have also incorporated active capabilities, with varying degrees of functionality. Some of the examples are Sybase [21], InterBase [38], INGRES (V6.0 and higher), and Rdb/VMS. These systems support rules that can be triggered on database events, viz. insert, delete and update, with the restriction that the event part can involve only one relation. Sybase further limits three rules (one for each database operation) per relation, and does not allow nesting of triggers, that is, one trigger cannot fire another. InterBase and commercial INGRES allow any number of triggers to be specified on a single relation. They also allows multiple rules to trigger on a single event, with user specified priority, and permits nested execution of triggers. In Rdb/VMS and InterBase, user can specify whether a rule should fire before or after a database operation. The major restriction in all of these systems is that triggers cannot be specified on derived or temporary data. Another restriction is that triggers are always

executed in immediate mode (in-line expansion of the action part if the event becomes true) as part of the triggering transaction.

In the rest of this chapter, we briefly describe the ongoing research in active databases, with special emphasis on the optimization techniques used by these systems for efficient situation evaluation. The goal of this survey is to draw useful results from existing systems, compare and contrast their approaches, find their limitations and investigate the possibilities for further improvements. For this purpose, we have chosen a set of basic requirements on which to base the comparison. These requirements include expressiveness, execution semantics, rule management, and optimization. We concentrate on optimization aspects of rule evaluation. A more detailed comparative study of expressiveness and execution semantics features can be found in recent literature [18, 17, 13].

2.1 Expressiveness

Active database rules have three parts: event, condition and action, as shown in Figure 2.1. The event expression specifies the system to register occurrence of the specified events. If the event-detector observes occurrence of the event, the corresponding condition part is evaluated. Event and condition parts together constitute a *situation* which can be written as $event \wedge condition$. If the situation evaluates to a true value then the corresponding action is evoked. These three parts of the rule specification provide maximum flexibility and expressibility, and avoid some of the problems in earlier work. ECA format of rules seems to be gaining consensus among researchers in the database community [32]. However, the three components have been grouped in a number of ways. Sybase [21] and InterBase [38] view rule as composed of an event part, and a condition part embedded inside the action part (as

II in Figure 2.1) which is like a transaction (a sequence of TRANSACT-SQL statements). ETM (Event Trigger Mechanism [40]) also follows the same philosophy but, in addition, supports non-database events, too. On the other hand, POSTGRES [58], Starburst [64], Ariel [34], HiPAC [11, 49] and Sentinel [16, 17] have separate event, condition and action parts (as III in Figure 2.1). The case I is generally followed by expert systems, e.g., OPS5 [30]. They just have a condition part and an action part.

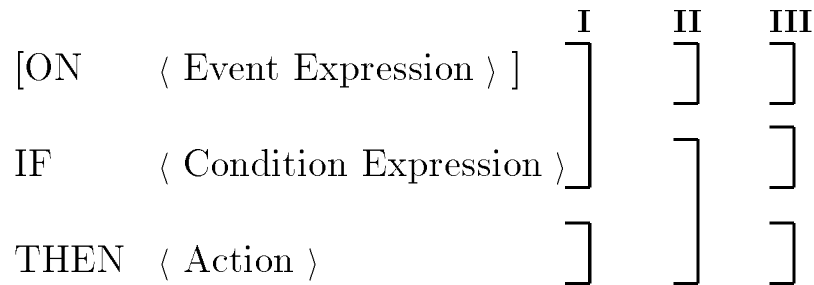

Figure 2.1. Rule Format Possibilities.

Active database systems require both database and external events to be expressed. All the above mentioned systems support database events but only HiPAC and ETM support external events. The need of temporal events has been recognized in HiPAC. POSTGRES and Ariel support a few specific temporal events (e.g. time and date.) In addition, for modeling the requirements of non-traditional applications, complex events are needed which translates to providing for composition of events. POSTGRES and Starburst support only disjunction type of event constructors. HiPAC provides event constructors viz. disjunction, sequence and closure. Sentinel supports a powerful event language, Snoop, which subsumes most of the features of other systems [17].

The condition part of a rule is a query. However, situation-action rules are normally triggered due to some change in the state of the database, which implies the pertinence of the deleted, modified (updated) and newly added tuples to conditions.

To accomplish this there exist two alternatives in the literature. First, extend the standard underlying query language to allow access to these tuples. Second, store these changes into some standard tables and allow user to access them in the same way as any other table.

Most of the systems follow the second alternative. Among the commercial DBMSs, Sybase allows access to the old and new values of tuples with the help of two logical tables: (*inserted* and *deleted*). InterBase uses the key words: *old* and *new*. These logical tables can be viewed as representing changes to a relation — as they store all the tuples inserted and/or deleted from a relation. Starburst and Ariel also follow the same philosophy. POSTGRES follows the first alternative. It extends the query language by allowing *new* and *old* to be tagged before tuple variables.

The above mentioned tables are useful when a rule is supposed to trigger due to a change in stored relations. However, they are inadequate for triggering a rule based on a change in derived relations (or, relational expressions). HiPAC supports a general structure, called *delta*, to capture changes to both stored and derived relations. Changes to stored relations are computed at the transaction (which cause the rule to triggers) execution time. For computing changes to derived relations *Changes* operator has been proposed. Similar philosophy is chosen for Sentinal [16] with minor differences, a few extensions and relaxing restrictions present in earlier work. Besides *Changes*, it supports incremental operators for all the relational operators which can be used inside conditions.

2.2 Execution Semantics

Semantics of rule execution determines how the condition part is evaluated and action is executed, after occurrence of an event. Most of the above mentioned system subscribe to the well-known match-resolve-act cycle algorithm of AI systems (e.g., OPS5 [30]), or some alteration of it. However, in contrast to AI systems, most of

the active database systems allow rules to trigger due to sets of changes: that is, condition part returns a set of tuples and passes on to the action part for execution of action. On the contrary, AI rules match a single fact in one cycle and execute the corresponding action.

HiPAC takes a very general approach to semantics of rule executions, following an extended nested transaction model [11, 45]. It supports a range of coupling modes between event detection and condition evaluation, and between condition and action:

- Immediate: immediately after the event and before the next operation in the triggering transaction.
- Deferred: after the last operation in the triggering transaction and before it commits.
- Detached: in separate transactions.

On the basis of these coupling modes, the following seven possibilities of rule execution are suggested:

1. Evaluate condition and execute action immediately after event occurs and before the next operation in the transaction.
2. Evaluate condition immediately after event occurs and execute action after the last operation in the transaction and before it commits.
3. Evaluate condition and execute action after the last operation in the transaction and before it commits.
4. Evaluate condition immediately after event occurs and execute action in a separate transaction.
5. Evaluate condition after the last operation in the transaction and execute the action in a separate transaction.

6. Evaluate condition and execute action together in a separate transaction.

7. Evaluate condition and execute action in two separate transactions.

Detached mode is further classified into two modes. First, *Causally Dependent*, meaning that condition is evaluated in a separate transaction but after the triggering transaction has committed. Second, *Causally Independent*, where condition is evaluated in a separate transaction which is independent of the triggering transaction

All the other systems support only a subset of the above coupling modes. Starburst supports deferred mode (like 3 above). Ariel supports both immediate and deferred modes (like 1 and 3 above). ETM, POSTGRES, Sybase and InterBase support only immediate coupling mode (like 1).

All the systems, except Sybase, allow cascaded execution of rules, meaning that one rule can trigger another rule. However most of them put a pre-specified limit on depth of cascading, for example, POSTGRES, Ariel, Starburst and commercial INGRES.

All the systems, except Sybase, allow triggering of multiple rules due to a single event. All the above systems give certain ordering to the potentially firable rules activated by an event. ETM, POSTGRES, Ariel, InterBase and commercial INGRES give user independence of specifying the order. Starburst uses a conflict resolution scheme similar to the ones used in expert systems. POSTGRES arbitrarily chooses one rule in case more than one triggered rules have the same priority. Ariel uses conflict resolution in such situations, similar to Starburst's. HiPAC interleaves multiple rule execution using serializability as a correctness criterion. Starburst and Ariel use an incrementally computed context for execution of the rules which were triggered by an operation before.

2.3 Management of Rules

The set of all *event-condition-action rules* is likely to form a potentially large set that need to be efficiently managed. Many of the systems support powerful rule management features. These features include creation, removal, activation and deactivation of rules and logical grouping of rules for efficient storage and aiding in evaluation. POSTGRES groups the rule set in an inheritance hierarchy. HiPAC takes a more general approach. It treats rules as first class objects, in the same way as data objects [22], allowing sharing of subparts, inheritance of properties and so on.

2.4 Situation Optimization

The set of all *event-condition-action rules* is likely to form a potentially large set of predefined queries that need to be efficiently evaluated when specified events occur. The ability of an active DBMS to evaluate complex situations efficiently is critical to the performance of an active database management system, since rule evaluation imposes an overhead on, possibly, every database update or any other event. In the case of time-constrained application requirements, for example in real-time systems, efficiency is of further more importance.

Optimization of a rule requires efficiently evaluating all the three parts of rule. That is, efficient detection of events, evaluation of conditions and execution of actions. Also, the way in which events, conditions, and actions are grouped together for execution with respect to their triggering transaction will also play a critical role in the selection of optimization techniques. If the entire rule is being optimized, subexpressions can be freely moved across the components during optimization. The knowledge of the action and event parts can be beneficially used for optimizing the condition part, rather than doing it in isolation. Also, as the number of rules is likely to be large, the ability to group related rules (rules that have common subexpressions

in events and conditions) to reduce computation is quite important. Efficient indexing techniques need to be applied to determine all the firable rules in a given situation.

The condition part of a rule is a query. So, it is easy to imagine that the general query optimization techniques can be applied in the case of condition expressions also. However, situation-action rules are normally triggered due to a change in database state, which implies that rules access two database states, and especially the *changes* (e.g., by inserting, deleting or updating tuples). On the other hand, queries just access a single state of the database. Conventional query processing techniques do not seem to be adequate for optimizing rules. Some of the extant systems have proposed useful extensions to query optimization and also some new techniques, including efficient indexing techniques.

In the rest of this section, we describe the rule optimization techniques used by some of the prominent database. As some of the systems, for example Starburst, use production system approach, it can be assumed that they exploit techniques similar to RETE and TREAT, however they do not mention explicitly about optimization. That is why we have included AI systems also in our study.

2.4.1 Optimization in ETM

ETM supports fast access paths to events, actions and triggers. It maintains a hash table for each which are kept in the main memory during run time. Pointers are used to speed up activation and deactivation of triggers. It uses a scope concept to restrict the size of main memory according to the local environment of the object currently being worked on. Executable code of actions is stored in a so called action-pool while source code is stored in a source-pool. Each event-table entry has a list of triggers currently associated with its event. Action is run as a child process communicating with the DBMS via Unix pipes.

It has been claimed that ETM rules are only marginally costly compared to the 'hardwired' solution, but provided far more benefits of flexibility.

2.4.2 Optimization Techniques in HiPAC

HiPAC's *Rule Evaluation Subsystem* (RES) is responsible for efficiently managing rules and evaluating complex situations. The RES maintains a collection of *signal graphs*. Each of these graphs represents a set of conditions with shared subparts. When a signal is received, appropriate graphs are evaluated to find out the actions to be fired. So, basic function of signal graphs is situation (event-condition pair) evaluation. HiPAC provides *Situation Evaluation Subsystem* (SES) for this. A signal graph is an operator graph in which nodes represent a relation and leaves are bound to events or to a database relation. The following three strategies have been proposed for evaluation of signal graphs.

- Pure bottom-up evaluation
- Top-down evaluation
- Signal driven approach

Signal driven approach is a combination of data and command driven evaluation. In addition to this, techniques have been proposed for transforming multiple situations on a signal graph. These techniques take advantage of the common subexpressions in a set of situations and generates an optimal signal graph for the whole set.

HiPAC situation evaluation has the following four steps.

1. *Parsing*
2. *High-level transformations*
3. *General-purpose optimizations*

4. *Interpretation*

In the *Parsing* phase situations are translated into signal graphs. Transformations specific to situation evaluation are performed in the *High-level transformations* phase. Conventional query optimization, such as permuting selections and joins, selecting an implementation for each join and index usage, is performed in the *General-purpose optimizations* phase. *Interpretation* is the final phase in which received signals are optimized and the signal graph is traversed to find out the root (output) node. Several techniques have been proposed for the optimization of different combinations of situations.

HiPAC has proposed use of *Changes* operator in the condition expressions. *Changes* is an operator that evaluates changes to a relational expression. The notion of *delta-relations* has also been developed which, in conjunction with *Changes*, allows encryption of changes to select-project-join (SJP) expressions. A *Chain Rule* has also been proposed for transforming *Changes* expressions into the expressions with Incremental Operators. *Changes* and incremental operators allows incremental rule evaluation. It was found that HiPAC's incremental operators and chain rule are very restrictive, and also have some inconsistencies. These are briefly summarized in Appendix A. We also show how our approach overcomes those problems, and generalizes and extends the earlier work.

For optimizing the condition part, HiPAC proposes using multiple query optimization techniques with the following rationale. Rules are *temporally persistent*, that is, they have a longer life-span and results are likely to be evaluated many times. This suggests that several rules can be optimized simultaneously in a group, using some of the techniques developed for multiple query optimization [29, 14, 54, 10]. The effect of multiple query optimization is proposed to be further enhanced by materializing intermediate results (e.g., common subexpressions).

For real-time and timing-constrained applications, HiPAC proposes a range of additional optimization techniques, such as exhaustive optimization, novel buffering strategies, use of main memory, and appropriate processing techniques (e.g., use of parallelism).

2.4.3 POSTGRES (PRSII) Optimization Techniques

POSTGRES applies different alternatives for rule optimization. These include indexing, cashing, lazy-evaluation and eager-evaluation. It supports two methods of rule processing: tuple level processing (called when individual tuples are deleted, inserted, modified or accessed) and query rewrite processing (converts user command to an alternative form prior to optimization.) The latter one is used to process more efficiently a subset of rules. The first is through the tuple level processing which is called when individual tuples are updated. The second is the query rewrite implementation which resides between the parser and the query optimizer and converts a user command to an alternate form which is more efficient.

Tuple Level Rule System: In this case when the executor processes a define rule command, information about the rule is inserted in system's catalogs and rule locks (special persistent markers) are placed at either the relation or the tuple level. POSTGRES uses the standard query optimizer to process situations. If the generated access plan uses index scan, then locks are placed on all the tuples read during the scan and on all the index intervals inspected. If the generated access plan uses a sequential search then a relation-level lock, termed as *lock escalation*, is placed on the relation being scanned.

There are three types of rule locks, event, import and export locks. Event locks are placed on appropriate fields in all the tuples of the relation that appears in the event specification of a rule and satisfy the event qualification.

For example:

```
define rule Ex3
on retrieve to EMP.salary
where EMP.age > 50 and EMP.dept = DEPT.dname
                      and DEPT.floor = 1
then do instead retrieve (salary = 10000)
```

Here, Events locked are placed on the salary field of all employees who are more than 50 years old and work in a first floor department. Event locks are tagged with the type of event which will activate the action along with the identifier of the rule. When this event occurs, the condition is evaluated and if true, the action is executed.

It is possible that event lock may not remain in the right tuples during updates. For example if a department is transferred to first floor, we have to add locks to all the employees working there. Use of two additional lock, Export and Import, keep the event locks correct as updates occur [56].

Query Rewrite Implementation: This module exists between the parser and the query optimizer and converts an incoming POSTGRES command, to which a rule applied, into an equivalent but efficient form, prior to optimization. For example consider the following rule:

```
define rule Ex5
on retrieve to TOY_EMP
then do instead retrieve(EMP_OID = EMP.OID, EMP.name,
                              EMP.age,EMP.salary)
EMP.dept = “toy”
```

and incoming query is

retrieve(TOY_EMP.salary) where TOY_EMP.name = "Sam"

Here, tuple level implementation will materialize all the tuples in TOY_EMP in order to find Sam's salary, while query rewrite implementation will convert the query into following equivalent form:

retrieve(EMP.salary) where EMP.name = "Sam" and
EMP.dept = "toy"

which is more efficient.

The locking strategy of POSTGRES has an advantage over the AI techniques that no main-memory is required to store index information of situations, rather this information is stored in the secondary storage with tuples and relations. This allows large number of rules to be supported. Another advantage is that it allows standard query optimization techniques to be used.

2.4.4 Ariel Rule Optimization

It has been realized in Ariel that in selecting an execution plan for actions, statistical information about some of the arguments may be unknown until the run time. This information is important because it determines what kind of access strategy would be optimal. For example, if the size of arguments is small nested-loop join can be used, but when size to too large sort-merge join would be more efficient. Ariel has proposed four approaches to this problem [34].

In the first approach, *never optimize*, an assumption is made about the size of the arguments and execution plan is generated and stored, all at compile time. It has been proposed that several plans (for different size assumptions) can be stored and suitable one is chosen at run time. Advantage of this approach is low overheads at run time.

The second approach, *Heuristic*, is similar to the last one but it tests for cost viability of the stored plan(s). If no stored plan is close to the actual cost, a new plan is computed. Otherwise one of the stored plans is used.

Third approach, *Cache and Invalidate*, is also similar to the first one, in that several plans are generated and stored at compile time. However, it allows generation of new plans and invalidation of old ones depending on significant changes in the size or data distribution of the stored relations, with help of a daemon process.

In the final approach, *Always Optimize*, a new strategy is generated everytime a rule triggers. It risks high overheads of compilation at run time, but always produces the most optimal plan.

2.4.5 Optimization in AI Systems

Rules in AI systems, called production rules, have two parts: condition and action. Event part is implicit. RETE [30] and TREAT [43] are two most prevalent algorithms used in these systems to evaluate the condition part.

These algorithms make use of the *temporal redundancy*, meaning that after a transaction only a very small fraction of the database (factbase) changes. They construct *discrimination networks* to filter out the data objects (tuples) which satisfy the condition part. RETE network has α-memories containing all objects that match a particular selection conjunct of a condition, and β-memories containing combinations of objects that match a join condition. Both of them are stored in main memory. TREAT networks do not have β-memories, instead they regenerate the join results whenever needed.

The main criticism to these algorithms is that they are main-memory based which restricts there use in large rule-bases, especially needed in database rule systems. To avoid this problem, virtual memory nodes have been suggested in the literature [36].

CHAPTER 3
PROBLEM STATEMENT AND APPROACH

3.1 Motivation

Motivation for this work comes from realizing that rules normally trigger due to database state transitions which requires evaluating changes, not only to tuples and relations (as in most of the extant systems), but also to derived relations (views). Further, because situation evaluation is critical to the performance of a database system, evaluation of changes is required to be done in an optimal way, as much as possible.

As we looked in the survey, there is a lack of a general framework for computation of changes to arbitrary relational expressions, even when most of the extant systems realize the importance of changes. Optimization of situations in the presence of changes was also found not having been investigated fully. Furthermore, evaluation of changes and rule triggering in distributed environments is not addressed adequately in the literature.

3.2 Requirements

An active database requires rules to trigger due to stored as well as derived data. This necessitates the need of a general purpose mechanism to compute changes. In the case of distributed relations and rules over different sites, this mechanism should permit evaluation of changes to such data also.

As we saw in the literature survey, many systems allow accessing logical tables which store changes at relational level. These tables should be utilized, as much as

possible, to compute changes to arbitrary expressions. Reason for this is that size of these tables is normally very small.

3.3 Approach

There exist two approaches for evaluating changes to an expression. In the first approach, extensions (or, snapshots) of the expression are computed both before and after the triggering transaction. Then, make use of the standard Set Difference to evaluate changes to the expression. This approach is very expensive as it requires recomputation of the expression twice, and also requires using Set Difference operator on potentially large relations, which makes it very costly for evaluating changes.

In the other approach, changes to expressions are evaluated in an incremental (or, differential) manner. To satisfy this end, we need a general purpose mechanism to evaluate changes to any relational expression by using changes to the involved stored relations. This thesis besides providing an algebraic framework for accomplishing this, also proposes ways of optimizing condition-expressions involving changes.

We propose *Incremental Operators* for evaluating changes to simple Select, Project, Join, Union and Set Difference operators. Changes to arbitrary non-aggregates can be evaluated using these incremental operators and relational operators. To achieve this, we have proposed the *Generic Chain Rule*. Two approaches have been proposed for optimization, and equivalence transformations on the incremental operators have been proposed.

CHAPTER 4
SENTINEL EXTENDED RELATIONAL ALGEBRA

Sentinel **E**xtended **R**elational **A**lgebra (*ERA*) provides a mathematical framework for optimizing situations in active databases. ERA extends the relational algebra with a capacity of expressing and optimizing conditions involving changes. The basic operators of this algebra are: *IncrSelect*, *IncrProject*, *IncrJoin*, *IncrUnion* and *IncrDiff* (for Set Difference). These operators are called *Incremental Operators* as the evaluate changes to the expressions with corresponding relational operators.

Using the incremental operators and relational operators, changes to arbitrary derived relations (non-aggregate relational expressions) can be expressed in a graph form. This allows application of optimizing transformations, on both relational and incremental operators to evaluate changes in an efficient way. This will be discussed in Section 3, in detail.

In this section, we define incremental operators. An incremental operator (say, *IncrOp*) evaluates changes to a derived relation obtained by applying the relational operator *Op* to its arguments when one or more arguments of *Op* change due to a transaction[1]. For example, *IncrSelect* will generate all the changes (deletions, insertions and updates) to *Select(R)* when R is changed due to a transaction, say T.

4.1 Basic Definitions

Before defining the operators of our algebra, we shall first present some basic definitions (of *delta*, *Compose* and *Changes*) which will be used later in the discussion.

[1] Without loss of generality all the concepts presented in this paper are equally applicable to any logical block (sequence) of database operations. Transaction is a special case which is used here for concreteness.

A *delta* corresponds to changes in the corresponding stored or derived relation. Delta of a stored relation is generated at the transaction execution time. *Changes* is an operator which evaluates changes to a derived relation by using deltas of the stored relations. It stores the output into another delta relation. *Compose* is an operator which evaluates the new extension of a relation, given the corresponding delta and the old relation extension.

<u>Delta:</u> *Delta* captures all the inserted, deleted and updated (with both old and updated values) tuples corresponding to a stored relation or a derived relation (relational expression). The delta of a relation R, represented by $\triangle R$, is a short-hand for two independent relations d_R and i_R which are defined on the same schema as that of R.[2] d_R stores all the deleted tuples and the old values of the updated tuples in the relation R. Similarly i_R stores all the inserted tuples and the new values of the updated tuples.

The following assumptions are made in the above definitions of d_R and i_R: **1.** $d_R \subseteq R$, **2.** $i_R \cap R = \phi$, and **3.** the TIDs (tuple IDs) of the old and new values of an updated tuple are equal. From 1 and 2, it follows that $d_R \cap i_R = \phi$. These assumptions provide net changes as inserting the same tuple after its deletion is equivalent to not changing anything. And, TIDs are supposed to be same in the assumption 3 because the tuple is still the same, logically.

An example of delta is given in Figure 2. Effect of the transaction T (in **a**) on EMP (in **b**) is captured in d_EMP and i_EMP (in **d**). Note $\triangle EMP$ is just a short-hand for the two relations d_EMP and i_EMP. EMP' shows the extension of EMP after the transaction T is committed.

[2] Note that our delta is just a notation, while HiPAC's [49] delta is a relation on a new schema. HiPAC's delta can be obtained by taking an OuterJoin of our d_R and i_R. Comparison of the two approaches is given in Appendix A.

Definition of Compose: If $\triangle R$ is the effect of a logical block of operations (eg, a transaction) on R, then the new extension of R (represented by R'), after execution of the block, is given by

$$R' \equiv Compose(R, \triangle R) \stackrel{\text{def}}{=} (R - d_R) \cup i_R \tag{4.1}$$

Definition of Changes: Let $V = E(R_1, \ldots, R_n)$ be a relational expression defined on the relations $R_1, \ldots, R_n$ where each R_i denotes an arbitrary relation; Let $\triangle R_i$ denote a delta of R_i and $R'_i = Compose(R_i, \triangle R_i)$, then $\triangle V$ will have two components d_V and i_V given by

$$Changes(V, [\triangle R_1, \ldots, \triangle R_n]) \stackrel{\text{def}}{=} \triangle V \tag{4.2}$$

where

$$\begin{aligned} d_V &= E(R_1, \ldots, R_n) - E(R'_1, \ldots, R'_n) \\ i_V &= E(R'_1, \ldots, R'_n) - E(R_1, \ldots, R_n) \end{aligned}$$

We call this method as evaluation of *Changes* by *First Principle*. This method is very inefficient as it requires recomputation of the whole expression (E) after the transaction, in addition to computing it before. It also requires use of two Set Difference operators, on probably big relation (as E can be any complex relational expression), which is expensive.

We propose another way of evaluating changes to derived relations which is much more efficient than the recomputation strategy. In this, we transform the *Changes* operator into an operator graph having the standard relational operators and the *Incremental Operators*. Changes to any arbitrary non-aggregate relational expression can be expressed using only these two type of operators. Thus avoiding use of the expensive *Changes* operator. This will become clear when we discuss the *Generic Chain Rule* and *Incremental Transformations* in the next section. In the rest of this section we present Incremental Operators of our algebra.

4.2 Operators of ERA

4.2.1 IncrSelect

IncrSelect evaluates changes to $V = Select_{pred}(R)$ when R is changed due to a transaction (or a block) and the effect of the transaction is stored in $\triangle R$ (ie, d_R and i_R). This operator provides an efficient way of evaluating changes to a Select expression. Our definition of IncrSelect is similar to HiPAC's [11, 49] if the OuterJoin in their definition is removed. It is also similar to what is proposed in the incremental view update context [4].

Definition of IncrSelect:

$$Changes(Select_{pred}R, [\triangle R]) \equiv IncrSelect_{pred}(\triangle R) \stackrel{\text{def}}{=} \triangle V \tag{4.3}$$

where

$$\begin{aligned} d_V &= Select_{pred}(d_R) \\ i_V &= Select_{pred}(i_R) \end{aligned}$$

Proof of IncrSelect:

$$\begin{aligned} V &\Leftrightarrow Select_{pred}(R) \\ &\Leftrightarrow Select_{pred}((R - d_R) \cup d_R) \\ &\Leftrightarrow [\overbrace{Select_{pred}(R) - Select_{pred}(d_R)}^{a}] \cup \overbrace{Select_{pred}(d_R)}^{b} \ldots (I) \\ V' &\Leftrightarrow Select_{pred}(R') \\ &\Leftrightarrow Select_{pred}(R - d_R \cup i_R) \\ &\Leftrightarrow [\overbrace{Select_{pred}(R) - Select_{pred}(d_R)}^{a}] \cup \overbrace{Select_{pred}(i_R)}^{c} \ldots (II) \end{aligned}$$

In I and II, a, b and c are exclusive sets because $i_R \cap R = \phi \Rightarrow Select_{pred}(i_R) \cap Select_{pred}(R) = \phi$, and in I, obviously, set a does not contain any elements of set b.

So, subtracting II from I, we shall get d_V and subtracting I from II, we get i_V, as follows:

$$d_V \equiv (V - V') \quad \Leftrightarrow \quad Select_{pred}(d_R)$$

$$i_V \equiv (V' - V) \quad \Leftrightarrow \quad Select_{pred}(i_R)$$

Example 1: Say, $\triangle V = IncrSelect_{R_1_value \geq 200}(\triangle R_1)$, then evaluate $\triangle V$, given R_1 and $\triangle R_1$ as follows:

R_1	
R_1_tid	R_1_value
@01	799
@02	200
@03	179

$\triangle R_1$			
d_R_1		i_R_1	
R_1_tid	R_1_value	R_1_tid	R_1_value
@01	799		
@03	179	@03	700
		@04	098

Applying the above definition of IncrSelect, we get

$\triangle V$			
d_V		i_V	
V_tid	V_value	V_tid	V_value
@01	799		
		@03	700

$R_1' = Compose(R_1, \triangle R_1)$ is

R_1'	
$R_1'_tid$	$R_1'_value$
@02	200
@03	700
@04	098

Now if we apply the First Principle (equation 4.2) and use R_1 and R'_1, we get the same result as $\triangle V$ in the above table.

4.2.2 IncrProject

HiPAC's [11, 49] definition of IncrProject seems to assume that the projected set of attributes always includes the key of the relation which means that duplicates are not allowed in the projected view (details in Appendix). Ceri and Widom [9] also have the same restriction. Blakely et al. [4] have proposed a method which removes this restriction, but requires storage of duplicate count. We are providing an alternative approach that does not require the duplicate count, but uses the difference operator on bags instead of sets. It is also general for all cases of insert/delete/update.

Blakely et al. [4] have proposed two ways of updating project views (which can be considered as alternatives for evaluating IncrProject). One alternative requires all the key attributes to be included in every project view (to disallow duplicates) which makes it very restrictive. The other alternative, without the key restriction, requires the duplicate count of tuples in the projected view to be stored in an extra column added to every projected view. To do this, Blakely et al. have re-defined the Project operator, which has complex semantics. Although not mentioned explicitly [4], the Set Difference operator also needs to be redefined to take care of the duplicate count. Also, presence of an extra column in the view may add other complexities, for example, whenever a project view is needed, user/system will have to remove that extra column. Besides, the definition, as presented in their paper, takes care of deletions, only. Needless to mention that in a project view, there can be cases when n tuples of the same value are added and m tuples of the very same value deleted. Our definition of IncrProject is general enough to handle all cases of deletes/inserts/updates (see Example 2). Moreover, we do not need an extra column to store duplicate count, which allows us to treat IncrProject in the same manner as any other operator. There

may be cases where the definition given by Blakely et al. might create complexities in evaluating changes to complex expressions involving multiple Selects, Projects and Joins.

Assume $V = Project_A(R)$, and $\triangle R$ (d_R and i_R) be the delta of R. A is a set of attributes of R. Note that A cannot have TID as it is not visible to the user. TID of the projected relation will be assigned by the system. Then $\triangle V$ can be efficiently evaluated using the following definition of IncrProject:

Definition of IncrProject:

$$Changes(Project_A R, [\triangle R]) \equiv IncrProject_A(R, \triangle R) \stackrel{\text{def}}{=} \triangle V \quad (4.4)$$

where

$$\begin{aligned} d_V &= ((\overbrace{\Pi^+_A(d_R) - \Pi^+_A(i_R)}^{temp}) \uplus \delta(temp)) - \Pi^+_A(R) \\ i_V &= \delta(\Pi^+_A(i_R) - \Pi^+_A(d_R)) - \Pi^+_A(R) \end{aligned}$$

Here, we have used the bag definition [1, 24] of Set Difference $(A - B)$, that is, if there are n tuple of one same value in A and m tuples of the same value in B then A will be left up with $n - m$ (0 if $n - m$ is negative) tuples of that value. This definition is upward compatible with the old definition, that is, it is as well applicable to the cases with no duplicates. The operator $\uplus$ just concatenates (or appends) the two sets together. The function δ removes all the duplicates from a set leaving behind only one copy. The operation Π^+ is like a regular Project but does not remove duplicates from the projected relation. Note that many of the existing systems already support bag definitions in their data models [8, 61, 51].

Proof of IncrProject:

This approach of evaluating $\triangle V$ (ie, d_V and i_V) is explained (and subsequently proved also) in an algorithmic way in Figure 4.2. Here, Step 1 is self-explanatory. We project (using Π^+) a set of attributes, represented by A (where, $TID \notin A$). Note that we cannot project the TID column because it is not visible to the user and it is identification for the whole set all attributes in the relation, not for just a subset.

Step 2 *compresses* the delta, that is, removes those tuples from i_R which are present in d_R also, and vice versa. This step is necessary because we don't want to consider these tuples which have been first inserted and then deleted later on.

To get i_V in the step 3, we remove duplicates (because even if n tuples of same value are inserted, we want to add only one to the view) from i2, and subtract $\Pi^+_A(R)$ (ie, R1) from it (because if a tuple of same value is already present in the view, we don't want to insert it).

Generating d_V in the step 3, is a little tricky. We know that if there are n tuples of the same value in $\Pi^+_A(R)$ (ie, R1) then d2 will have either

1. $< n$ tuples of the same value, or

2. $= n$ tuples of the same value.

The bag d2 cannot have more than n tuples (of the same value) because by the definition of delta one cannot delete a larger number of tuples than those are already present.

In the case 1 above, it is evident that the view does not change. But in the case 2, we want to have one (and only one) tuple of that value in d_V (this is because all the instances of that value have been deleted). Both of the cases can be satisfied by simply adding one extra tuple for every distinct value in d2 and then subtract $\Pi^+_A(R)$ (ie, $R1$) from it.

Example 2: Say, $\triangle V = IncrProject_{val1}(R, \triangle R)$, then evaluate $\triangle V$, given R and $\triangle R$, as follows:

R		
R_tid	R_val1	R_val2
@01	388	201
@02	388	537
@03	400	739
@04	999	200

$\triangle R$						
d_R				i_R		
R_tid	R_val1	R_val2		R_tid	R_val1	R_val2
@01	388	201		@01	400	235
@02	388	537		@02	700	321
@04	999	200				
				@05	388	526
				@06	400	593

After Step 1 (Figure 4.2), we get

$\Pi^{+}_{val1}(R)$
R_val1
388
388
400
999

and,

$\triangle temp$		
$d1$		$i1$
R_val1		R_val1
388		400
388		700
999		
		388
		400

After Step 2, we get

$\triangle temp$	
$d2$	$i2$
R_val1	R_val1
388	400
	700
999	
	400

After Step 3, we get

$\triangle V$	
d_V	i_V
R_val1	R_val1
	700
999	

This result can be confirmed by applying the First Principle (equation 4.2).

4.2.3 IncrJoin

IncrJoin is an operator which computes changes to a Join of two relations (say, $R_1 \bowtie R_2$) when either one or both of the relations change. Changes to a Join of arbitrary number of relations can be computed using this basic definition of IncrJoin (details in Chapter 5).

HiPAC's [11, 49] definition of IncrJoin produces spurious tuples when both of the participating relations change (see Appendix). However, it computes the changes correctly if only one of the participating relations change, or when there are no deletions. Blakely et al. [4] have also proposed a way for computing changes to join views but has similar problems. Hanson's definition [33] rectifies the problems present in the definition given by Blakely et al. and works well for all the cases. Our definition is quite similar to their definition, but is simpler and has less number of Joins.

Terminology: We shall use Figure 4.3 for describing IncrJoin. We use the shorthands d_1 and i_1 for d_R_1 and i_R_1 (which are parts of $\triangle R_1$,) respectively. r_1 is that part of R_1 which is left after removing deleted tuples (ie, $r_1 = R_1 - d_1$). d_2, i_2 and r_2 are defined similarly.

TID of the tuples in a Join of two relations can be obtained by concatenating the TIDs of the tuples in the two operand relations, as it will satisfy the uniqueness criterion. Although a unique system generated TID can also be assigned.

Definition of IncrJoin:

$$Changes(R_1 \bowtie R_2, [\triangle R_1, \triangle R_2]) \equiv IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2) \stackrel{\text{def}}{=} \triangle V \qquad (4.5)$$

where

$$\begin{aligned} d_V &= r_1 \bowtie d_2 \cup d_1 \bowtie R_2 \\ i_V &= r_1 \bowtie i_2 \cup i_1 \bowtie r_2 \cup i_1 \bowtie i_2 \end{aligned}$$

Proof of IncrJoin:

$$\begin{aligned} V &\Leftrightarrow (r_1 \cup d_1) \bowtie (r_2 \cup d_2) \\ &\Leftrightarrow r_1 \bowtie r_2 \cup r_1 \bowtie d_2 \cup d_1 \bowtie r_2 \cup d_1 \bowtie d_2 \ldots (I) \\ V' &\Leftrightarrow (r_1 \cup i_1) \bowtie (r_2 \cup i_2) \\ &\Leftrightarrow r_1 \bowtie r_2 \cup r_1 \bowtie i_2 \cup i_1 \bowtie r_2 \cup i_1 \bowtie i_2 \ldots (II) \end{aligned}$$

The only term common to the two equations (I and II) is $r_1 \bowtie r_2$. Other terms will give exclusive tuples since $i_1 \cap d_1 = \phi$ and $i_2 \cap d_2 = \phi$. So, we get the following terms in d_V by subtracting II from I, and in i_V by subtracting I from II:

$$d_V \equiv (V - V') \quad \Leftrightarrow \quad r_1 \bowtie d_2 \cup d_1 \bowtie r_2 \cup d_1 \bowtie d_2 \ldots (III)$$

$$\Leftrightarrow \quad r_1 \bowtie d_2 \cup d_1 \bowtie R_2$$

$$\Leftrightarrow \quad d_1 \bowtie r_2 \cup R_1 \bowtie d_2$$

$$i_V \equiv (V' - V) \quad \Leftrightarrow \quad r_1 \bowtie i_2 \cup i_1 \bowtie r_2 \cup i_1 \bowtie i_2$$

In III, $d_1 \bowtie r_2 \cup d_1 \bowtie d_2 \equiv d_1 \bowtie (r_2 \cup d_2) \equiv d_1 \bowtie R_2$, by the definition of delta and associativity of Join over Union.

Example 3: Let us assume that we are given R_1 and $\triangle R_1$ of the Example 1 and the following R_2 and $\triangle R_2$, then to evaluate $\triangle V$ where $V = R_1 \bowtie R_2$.

R_2	
R_2_tid	R_2_value
@10	799
@20	179
@30	200
@40	700
@50	098

and

$\triangle R_2$			
d_R_2		i_R_2	
R_2_tid	R_2_value	R_2_tid	R_2_value
@20	179		
@30	200	@30	645
@50	098	@50	503

Applying the above definition of IncrJoin, we get

$\triangle V$					
d_V			i_V		
R_1_tid, R_2_tid	R_1_value	R_2_value	R_1_tid, R_2_tid	R_1_value	R_2_value
@02,20	200	179			
@02,30	200	200	@02,30	200	645
@02,50	200	098	@02,50	200	503
@01,10	799	799			
@01,20	799	179			
@01,30	799	200			
@01,40	799	700			
@01,50	799	098			
@03,10	179	799	@03,10	700	799
@03,20	179	179			
@03,30	179	200	@03,30	700	645
@03,40	179	700	@03,40	700	700
@03,50	179	098	@03,50	700	503
			@04,10	098	799
			@04,30	098	645
			@04,40	098	700
			@04,50	098	503

This result can be confirmed by applying the First Principle (eqnuation 4.2).

4.2.4 IncrUnion

We define a new operator IncrUnion which computes changes to a Union of two relations (say, $R_1 \cup R_2$) when either one or both of the relations change. Changes to a Union of arbitrary number of relation can be computed using this basic definition of IncrUnion (details in Chapter 5). This operator is very useful in the case of horizontally distributed relations.

We assume that TID of tuples in the Union of two relations will come from concatenating TIDs of the tuples in the operand relations if the tuple is common in both. Otherwise, it will be the TID of the tuple in the corresponding relation.

Definition of IncrUnion

$$Changes(R_1 \cup R_2, [\triangle R_1, \triangle R_2]) \equiv IncrUnion(R_1, R_2, \triangle R_1, \triangle R_2) \stackrel{\text{def}}{=} \triangle V \quad (4.6)$$

where

$$\begin{aligned} d_V &= (d_1 - (d_1 \cap i_2) - (r_2 \cap d_1)) \cup (d_2 - (d_2 \cap i_1) - (r_1 \cap d_2)) \\ i_V &= (i_1 - (i_1 \cap R_2)) \cup (i_2 - (i_2 \cap R_1)) \end{aligned}$$

Proof of IncrUnion:

We shall use Figure 4.4 in this proof. In this, the two filled circles show the common tuples between R_2 and i_1, represented by $f1 + f2$. Similarly the empty circles represented by $e1 + e2$. Let us give these notations to the following regions in the figure:

$r'_1 \equiv \triangle bci$,

$d'_1 \equiv \Box biga$, and

$i'_1 \equiv \triangle agm$.

In the following proof, we shall use the regions (P, Q, Z, Y, W, e1, e2, f1 and f2) as marked in figure 4.4, also.

$$\begin{aligned} V &\Leftrightarrow R_1 \cup R_2 \\ &\Leftrightarrow (r_1 \cup d_1) \cup (r_2 \cup d_2) \\ &\Leftrightarrow (r_1 \cup r_2) \cup (d_1 \cup d_2) \\ &\Leftrightarrow P \cup Q \cup W \cup Z \cup e1 \cup e2 \cup f1 \cup f2 \cup (r'_1 - e1) \cup (r'_2 - f1) \cup (d'_1 - e2) \cup \\ &\qquad (d'_2 - f2) \ldots (I) \end{aligned}$$

$$\begin{aligned} V' &\Leftrightarrow R'_1 \cup R'_2 \\ &\Leftrightarrow (r_1 \cup i_1) \cup (r_2 \cup i_2) \\ &\Leftrightarrow (r_1 \cup r_2) \cup (i_1 \cup i_2) \\ &\Leftrightarrow Y \cup W \cup P \cup Q \cup e1 \cup e2 \cup f1 \cup f2 \cup (r'_1 - e1) \cup (r'_2 - f1) \cup \\ &\qquad (i'_1 - (f1 \cup f2)) \cup (i'_2 - (e1 \cup e2)) \ldots (II) \end{aligned}$$

Subtracting II from I, we get:

$$\begin{aligned} d_V \equiv (V - V') &\Leftrightarrow Z \cup (d'_1 - e2) \cup (d'_2 - f2) \\ &\Leftrightarrow (d_1 - e2 - Q) \cup (d_2 - f2 - P) \\ &\Leftrightarrow (d_1 - (d_1 \cap i_2) - (r_2 \cap d_1)) \cup (d_2 - (d_2 \cap i_1) - (r_1 \cap d_2)) \end{aligned}$$

Subtracting I from II, we get:

$$\begin{aligned} i_V \equiv (V' - V) &\Leftrightarrow Y \cup (i'_1 - (f1 \cup f2)) \cup (i'_2 - (e1 \cup e2)) \\ &\Leftrightarrow (i_1 - (f1 \cup f2)) \cup (i_2 - (e1 \cup e2)) \\ &\Leftrightarrow (i_1 - (i_1 \cap R_2)) \cup (i_2 - (i_2 \cap R_1)) \end{aligned}$$

Example 4: Let us compute $\triangle V$ of Union of the same two relations used in the last example ($V = R_1 \cup R_2$).

$\triangle V$				
d_V			i_V	
V_tid	V_value		V_tid	V_value
@03,20	179			
			@30	645
			@50	503

This result can be confirmed by applying the First Principle (eqnuation 4.2).

4.2.5 IncrDiff

IncrDiff (Incremental Difference) is introduced to compute changes to a Set Difference of two relations (say, $R_1 - R_2$) when either one or both of the relations change.

If R_1 and R_2 have $\triangle R_1$ and $\triangle R_2$ as their respective deltas, then IncrDiff of the two relation can be determined by using the equation given below. Terminology is the same as for the last two operators.

We assume that TIDs of tuples in the difference of two relations comes from the TIDs of the corresponding tuples of the first operand relation.

Definition of IncrDiff

$$Changes(R_1 - R_2, [\triangle R_1, \triangle R_2]) \equiv IncrDiff(R_1, R_2, \triangle R_1, \triangle R_2) \stackrel{\text{def}}{=} \triangle V \qquad (4.7)$$

where

$$\begin{aligned} d_V &= (r_1 \cap i_2) \cup (d_1 - (d_1 \cap R_2)) \\ i_V &= (i_1 - (i_1 \cap r_2) - (i_1 \cap i_2)) \cup (r_1 \cap d_2) \end{aligned}$$

Proof of IncrDiff:

In this proof, we use the same terms as defined for IncrUnion and Figure 4.4.

$$\begin{aligned} V &\Leftrightarrow R_1 - R_2 \\ &\Leftrightarrow r_1' \cup d_1' \\ &\Leftrightarrow (r_1' - e1) \cup (d_1' - e2) \cup e1 \cup e2 \ldots (I) \end{aligned}$$

$$\begin{aligned} V' &\Leftrightarrow R_1' - R_2' \\ &\Leftrightarrow (r_1 \cup i_1) - (r_2 \cup i_2) \\ &\Leftrightarrow [e1 \cup (r_1' - e1) \cup W \cup P \cup (i_1' - (f1 \cup f2)) \cup (f1 \cup f2) \cup Y] - \\ &\quad\ [f1 \cup (r_2' - f1) \cup W \cup Q \cup (i_2' - (e1 \cup e2)) \cup (e1 \cup e2) \cup Y] \\ &\Leftrightarrow (r_1' - e1) \cup P \cup f2 \cup (i_1' - (f1 \cup f2)) \end{aligned}$$

Now, subtracting II from I, we get d_V and by subtracting I from II, we get i_V, as in the definition above.

$$d_V \equiv (V - V') \quad \Leftrightarrow \quad (d_1' - e2) \cup e1 \cup e2$$

$$\Leftrightarrow \quad d_1' \cup e1$$

$$\Leftrightarrow \quad (r_1 \cap i_2) \cup (d_1 - (d_1 \cap R_2))$$

$$i_V \equiv (V' - V) \quad \Leftrightarrow \quad P \cup f2 \cup (i_1' - (f1 \cup f2))$$

$$\Leftrightarrow \quad P \cup (i_1' - f1)$$

$$\Leftrightarrow \quad (i_1 - (i_1 \cap r_2) - (i_1 \cap i_2)) \cup (r_1 \cap d_2)$$

Example 5: Let us compute $\triangle V$ on Set Difference of the same two relations used in the last example. Applying the above definition, we get

$\triangle V$			
d_V		i_V	
V_tid	V_value	V_tid	V_value
		@02	200
		@04	098

This result is the same as we shall get by applying the First Principle (equation 4.2).

Transaction T: Modify(@111, new_tuple = (@111, Mike, 45K));
Insert(@444, Sharon, 30K);
Delete(@333)

(a)

EMP	TID	Name	Salary
	@111	Mike	40K
	@222	Carol	35K
	@333	John	30K

(b)

EMP'	TID	Name	Salary
	@111	Mike	45K
	@222	Carol	35K
	@444	Sharon	30K

(c)

$\triangle$ EMP					
d_EMP			i_EMP		
TID	Name	Salary	TID	Name	Salary
@111	Mike	40K	@111	Mike	45K
@333	John	30K	–	–	–
–	–	–	@444	Sharon	30K

(d)

Figure 4.1. Structure of delta. (a) A transaction, T, on EMP. (b) A relation, EMP. (c) EMP after the transaction. (d) Delta relation of EMP.

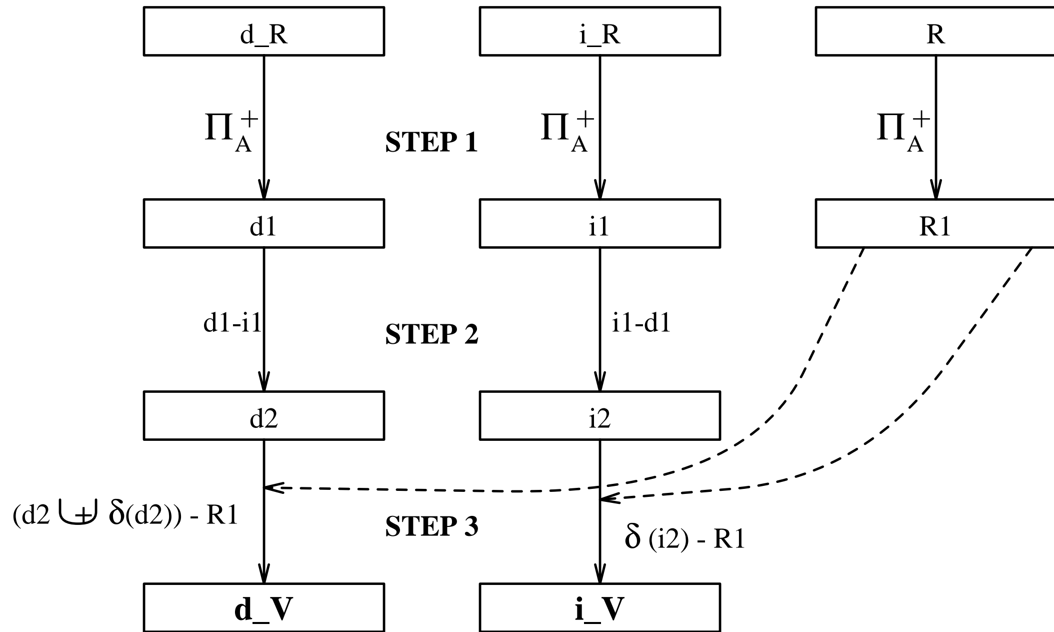

Figure 4.2. Algorithm for Evaluating Changes to a Project View.

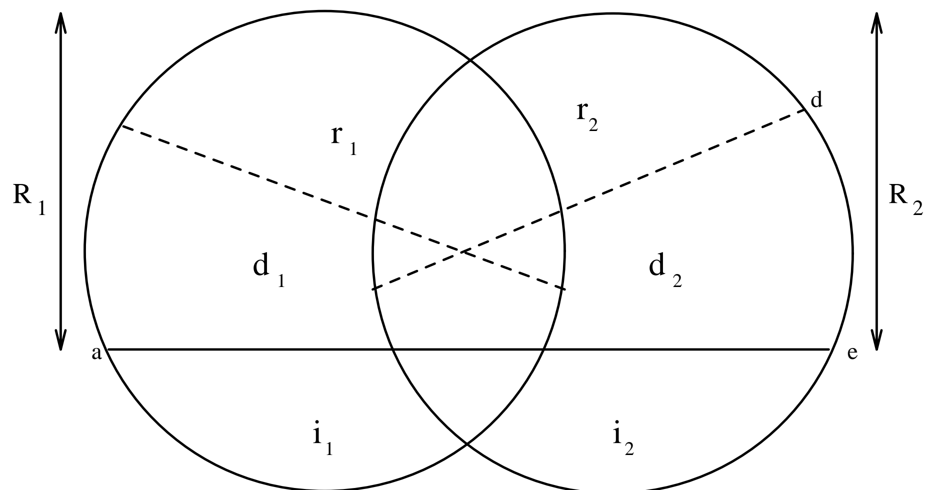

Figure 4.3. Shows overlap of deltas and relations for IncrJoin operator.

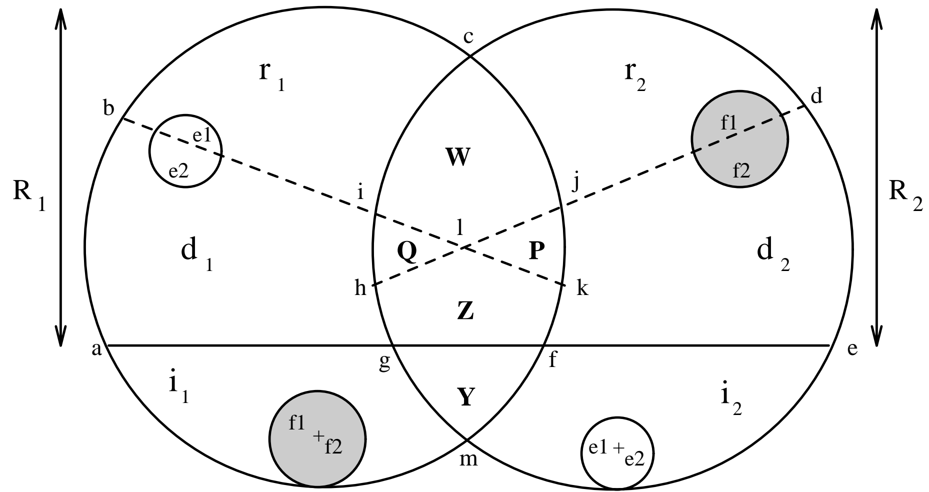

Figure 4.4. Shows overlap of deltas and relations for IncrUnion and IncrDiff operators.

CHAPTER 5
COMPUTING CHANGES TO COMPLEX EXPRESSIONS

In the last chapter we have showed how to efficiently evaluate changes to simple Select, Project, Join, Union and Set Difference expressions using the incremental operators. In this chapter, we propose two approaches to optimizing *Changes* expressions (Section 5.1). A generic chain rule is proposed for computing changes to complex relational expressions involving arbitrary number of the above operators (in Section 5.2). Optimization transformations over incremental operators are presented in Section 5.3 and their proofs are given in Section 5.4.

5.1 Optimization

Given an expression $Changes(RE, [\triangle RList])$ (where RE is an expression and $\triangle RList$ the list of deltas involved in RE), there are two approaches of optimizing it: **1.** use algebraic transformations on RE and then apply the Chain Rule (shown by the solid arrows in Figure 5.1), and **2.** apply the Chain Rule first and then use transformations on incremental operators (shown by the dashed arrows in Figure 5.1).

This first approach uses the chain rule and relational transformations. Let there be N possible equivalence transformations of the expression RE. We apply the chain rule to each of the N equivalent expressions and get the corresponding expression involving relational and incremental operators. From these expressions, we determine the best alternative. This approach requires N applications of the chain rule and an equal number of transformations.

The second approach uses the chain rule only once on $Changes(RE, [\triangle RList])$ and applies incremental transformations (see Section 5.3) on the result (of chain rule) to choose the best alternative. A criterion for applying the transformations to choose the best alternative is: pushing the incremental operators down the tree, as much as possible. This is helpful because incremental operators produce deltas which are smaller in size.

The second approach is more efficient as it requires only one application of the chain rule, compared with N in the first approach.

5.2 Generic Chain Rule

Generic chain rule allows changes to any arbitrary relational expressions to be expressed in a tree form involving only the above incremental operators and relational operators. This avoids the use of the expensive *Changes* operator, and more importantly gives opportunities for optimization. This chain rule is capable of handling arbitrary non-aggregate relational expressions. Furthermore it can handle any n-ary operator provided the incremental form of that operator exists. Currently, it is being extended for expressions involving aggregate operators. HiPAC's [11, 49] chain rule dealt with changes to only Select and Project expressions (details of comparison in Appendix).

The following equation recursively evaluates changes to an expression (RE) due to the changes to the individual relations (represented by $\triangle RList$,) present in the expression.

$$Changes(RE, [\triangle RList]) = \begin{cases} \triangle RE, & \text{(If RE is a stored relation)} \\ IncrOp(RE_1, RE_2, \ldots, RE_m, C_1, C_2, \ldots, C_n), & \\ \quad \text{(If RE is an expression with atleast one operator.)} & \end{cases}$$

where,

$$\begin{aligned} RE &= Op(RE_1, RE_2, \ldots, RE_n); \quad Op \text{ is an n-ary operator} \\ m &= \text{number of non-delta arguments of IncrOp; } m \leq n. \\ RE_i &= Op_i(RE_{i1}, RE_{i2}, \ldots, RE_{ip}) \quad Op_i \text{ is a p-ary operator} \\ C_i &= Changes(RE_i, [\triangle RList_i]) \\ & \quad [\text{where, } \triangle RList_i = \triangle RList \cap \triangle \text{ relations used by } IncrOp_i.] \end{aligned}$$

Proof: If RE is a stored relation then by the definition of delta *Changes* is equal to $\triangle RE$. If RE is a relational expression (say, $Op(\ldots)$, where Op is Select, Project, Join, Union or Difference), then by the definition of incremental operators, *Changes* can be written as Incr**Op**. The non-delta arguments of *IncrOp* (ie, $RE_1, \ldots, RE_n$) will directly come from the arguments of *Op*. The delta arguments of *IncrOp* (ie, $C_1, \ldots, C_n$) are the deltas corresponding to the relations derived from the arguments of *Op*. That is, C_1 is $\triangle RE_1$ which can be computed by applying the above chain rule again. Similarly, the other delta arguments of *IncrOp* can be computed.

This chain rule is illustrated in the following example.

Example 6: Let $\triangle RE = Changes(Select_{pred}(R) \cup ProjectA(S), [\triangle R, \triangle S])$ be the given expression as in Step 1 of Figure 5.2. Using the chain rule this can be transformed in an expression with IncrUnion (in Step 2). Here, the branches labeled a, b, $\triangle a$ and $\triangle b$ show the four arguments of IncrUnion. The first two (a and b) are obtained by applying the relational operators Select and Project on R and S, respectively. The other two arguments are obtained in Step 3 by applying the chain rule, again. $\triangle a$ represents changes to the expression $Select_{pred}(R)$, that is, $Changes(Select_{pred}(R), [\triangle R])$ and is transformed into an IncrSelect as shown in the figure. Similarly, we get $\triangle b$. The final output is:

$IncrUnion(Select_{pred}(R), Project_A(S), IncrSelect_{pred}(\triangle R), IncrProject_A(S, \triangle S))$.

5.3 Transformations of Incremental Operators

Following are the equivalence transformations of incremental operators:

1. Commutative law for IncrJoin:

$$IncrJoin_{pred}(R_1, R_2, \triangle R_1, \triangle R_2) \equiv IncrJoin_{pred}(R_2, R_1, \triangle R_2, \triangle R_1)$$

2. Associative law for IncrJoin:

$$IncrJoin(R_1, R_2 \bowtie R_3, \triangle R_1, IncrJoin(R_2, R_3, \triangle R_2, \triangle R_3)) \equiv$$
$$IncrJoin(R_1 \bowtie R_2, R_3, IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2), \triangle R_3)$$

3. Cascade of IncrProjects:

$$IncrProject_A(\Pi_B, IncrProject_B(R, \triangle R)) \equiv IncrProject_A(R, \triangle R)$$

This requires a sufficient and necessary condition (SNC) that $A \subseteq B$.

4. Cascade of IncrSelects:

$$IncrSelect_{p1}(IncrSelect_{p2}(\triangle R)) \equiv IncrSelect_{p1 \wedge p2}(\triangle R) \equiv$$
$$IncrSelect_{p2}(IncrSelect_{p1}(\triangle R))$$

5. Commuting IncrSelects and IncrProjects:

$$IncrProject_A(\sigma_p(R), IncrSelect_p(\triangle R)) \equiv IncrSelect_p(IncrProject_A(R, \triangle R))$$

It requires an SNC that the predicate p must involve only the attributes present in the set A.

6. Commuting IncrSelect and IncrJoin:

$$IncrJoin(\sigma_{p1}(R_1), \sigma_{p2}(R_2), IncrSelect_{p1}(\triangle R_1), IncrSelect_{p2}(\triangle R_2)) \equiv$$
$$IncrSelect_{p1 \wedge p2}(IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2))$$

Here, SNC is: $p1$ must access only the attributes of R_1 and $p2$ must access the attributes of R_2 only.

7. Commuting IncrSelect and IncrUnion:

$$IncrUnion(\sigma_p(R_1), \sigma_p(R_2), IncrSelect_p(\triangle R_1), IncrSelect_p(\triangle R_2)) \equiv IncrSelect_p(IncrUnion(R_1, R_2, \triangle R_1, \triangle R_2))$$

8. Commuting IncrSelect and IncrDiff:

$$IncrDiff(\sigma_p(R_1), \sigma_p(R_2), IncrSelect_p(\triangle R_1), IncrSelect_p(\triangle R_2)) \equiv IncrSelect_p(IncrDiff(R_1, R_2, \triangle R_1, \triangle R_2))$$

9. Commuting IncrProject and IncrJoin:

$$IncrJoin(\Pi_B(R_1), \Pi_C(R_2), IncrProject_B(R_1, \triangle R_1), IncrProject_C(R_2, \triangle R_2)) \equiv IncrProject_A(R_1 \bowtie R_2, IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2))$$

Here, the SNC is: B and C are disjoint sets of attributes of R_1 and R_2, respectively, and both are subsets of A.

10. Commuting IncrProject and IncrUnion:

$$IncrUnion(\Pi_A(R_1), \Pi_A(R_2), IncrProject_A(R_1, \triangle R_1), IncrProject_A(R_2, \triangle R_2)) \equiv IncrProject_A(R_1 \cup R_2, IncrUnion(R_1, R_2, \triangle R_1, \triangle R_2))$$

5.4 Proofs of Incremental Transformations

Proof of 1:

$$R_1 \bowtie R_2 \equiv R_2 \bowtie R_1$$

$$\Leftrightarrow \quad Changes(R_1 \bowtie R_2, [\triangle R_1, \triangle R_2] \equiv Changes(R_2 \bowtie R_1, [\triangle R_2, \triangle R_1]$$

$$\Leftrightarrow \quad IncrJoin_{pred}(R_1, R_2, \triangle R_1, \triangle R_2) \equiv IncrJoin_{pred}(R_2, R_1, \triangle R_2, \triangle R_1)$$

Proof of 2:

$$R_1 \bowtie (R_2 \bowtie R_3) \equiv (R_1 \bowtie R_2) \bowtie R_3$$

$$\Leftrightarrow \quad Changes(R_1 \bowtie (R_2 \bowtie R_3), [\triangle R_1, \triangle R_2, \triangle R_3]) \equiv$$

$$Changes((R_1 \bowtie R_2) \bowtie R_3, [\triangle R_1, \triangle R_2, \triangle R_3])$$

$$\Leftrightarrow \quad IncrJoin(R_1, R_2 \bowtie R_3, \triangle R_1, IncrJoin(R_2, R_3, \triangle R_2, \triangle R_3)) \equiv$$

$$IncrJoin(R_1 \bowtie R_2, R_3, IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2), \triangle R_3)$$

Proof of 3:

$$\Pi_A(\Pi_B(R)) \equiv \Pi_A(R); \quad A \subseteq B$$

$$\Leftrightarrow \quad Changes(\Pi_A(\Pi_B(R)), [\triangle R]) \equiv Changes(\Pi_A(R), [\triangle R])$$

$$\Leftrightarrow \quad IncrProject_A(\Pi_B(R), Changes(\Pi_B(R), [\triangle R])) \equiv IncrProject_A(R, \triangle R)$$

$$\Leftrightarrow \quad IncrProject_A(\Pi_B, IncrProject_B(R, \triangle R)) \equiv IncrProject_A(R, \triangle R)$$

Proof of 4:

$$\sigma_{p1}(\sigma_{p2}(R)) \equiv \sigma_{p1 \wedge p2}(R) \equiv \sigma_{p2}(\sigma_{p1}(R))$$

Applying the chain rule, we get:

$$\Leftrightarrow \quad IncrSelect_{p1}(IncrSelect_{p2}(\triangle R)) \equiv IncrSelect_{p1 \wedge p2}(\triangle R) \equiv$$

$$IncrSelect_{p2}(IncrSelect_{p1}(\triangle R))$$

Proof of 5:

$$\Pi_A(\sigma_p(R)) \equiv \sigma_p(\Pi_A(R))$$

Applying the chain rule, we get:

$$\Leftrightarrow \quad IncrProject_A(\sigma_p(R), IncrSelect_p(\triangle R)) \equiv IncrSelect_p(IncrProject_A(R, \triangle R))$$

Proof of 6:

$$\sigma_{p1}(R_1) \bowtie \sigma_{p2}(R_2) \equiv \sigma_{p1 \wedge p2}(R_1 \bowtie R_2)$$

Applying the chain rule, we get:

$$\Leftrightarrow \quad IncrJoin(\sigma_{p1}(R_1), \sigma_{p2}(R_2), IncrSelect_{p1}(\triangle R_1), IncrSelect_{p2}(\triangle R_2)) \equiv$$

$$IncrSelect_{p1 \wedge p2}(IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2))$$

Proof of 7:

$$\sigma_p(R_1) \cup \sigma_p(R_2) \equiv \sigma_p(R_1 \cup R_2)$$

Applying the chain rule, we get:

$$\Leftrightarrow \quad IncrUnion(\sigma_p(R_1), \sigma_p(R_2), IncrSelect_p(\triangle R_1), IncrSelect_p(\triangle R_2)) \equiv$$

$$IncrSelect_p(IncrUnion(R_1, R_2, \triangle R_1, \triangle R_2))$$

Proof of 8:

$$\sigma_p(R_1) - \sigma_p(R_2) \equiv \sigma_p(R_1 - R_2)$$

Applying the chain rule, we get:

$$\Leftrightarrow \quad IncrDiff(\sigma_p(R_1), \sigma_p(R_2), IncrSelect_p(\triangle R_1), IncrSelect_p(\triangle R_2)) \equiv$$

$$IncrSelect_p(IncrDiff(R_1, R_2, \triangle R_1, \triangle R_2))$$

Proof of 9:

$$\Pi_B(R_1) \bowtie \Pi_C(R_2) \equiv \Pi_A(R_1 \bowtie R_2)$$

Applying the chain rule, we get:

$$\Leftrightarrow \quad IncrJoin(\Pi_B(R_1), \Pi_C(R_2), IncrProject_B(R_1, \triangle R_1), IncrProject_C(R_2, \triangle R_2)) \equiv$$

$$IncrProject_A(R_1 \bowtie R_2, IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2))$$

Proof of 10:

$$\Pi_A(R_1) \cup \Pi_A(R_2) \equiv \Pi_A(R_1 \cup R_2)$$

Applying the chain rule, we get:

$$\Leftrightarrow \quad IncrUnion(\Pi_A(R_1), \Pi_A(R_2), IncrProject_A(R_1, \triangle R_1), IncrProject_A(R_2, \triangle R_2)) \equiv$$

$$IncrProject_A(R_1 \cup R_2, IncrUnion(R_1, R_2, \triangle R_1, \triangle R_2))$$

5.5 Example

Example 7: This examples illustrates the two approches of optimization presented in Section 5.1. Let $Changes(Select_{p1 \wedge p2}(R \bowtie S), [\triangle R, \triangle S])$ be the expression to be optimized. The solid arrows in Figure 5.3 show the path taken when applying the first approach, and the dashed arrows show the second approach.

In Figure 5.3, We have the following expression to be optimized on the corner A of the box:

$Changes(\sigma_{p1 \wedge p2}(R \bowtie S), [\triangle R, \triangle S])$

Following the second approach, we apply the chain rule to this expression and get

$IncrSelect(T, \triangle T) \equiv IncrSelect(R \bowtie S, IncrJoin(R, S, \triangle R, \triangle S))$

on the corner B. Now we use the incremental transformation #6 on this to get the final optimized expression:

$IncrJoin(\sigma_{p1}(R), \sigma_{p2}(S), IncrSelect(R, \triangle R), IncrSelect(S, \triangle S)) \ldots (I)$

The same optimized expression can be achieved by following the first approach. We first apply relational transforamations on the expression inside *Changes* on the corner A to get

$\sigma_{p1}(R) \bowtie \sigma_{p2}(S)$.

We assume that the predicate $p1$ is defined on the attributes of R only and $p2$ on those of S only. So, we get the following expression on the corner C

$Changes(\sigma_{p1}(R) \bowtie \sigma_{p2}(S), [\triangle R, \triangle S])$.

Now by applying the chain rule, we get the same optimized expression as in the equation I.

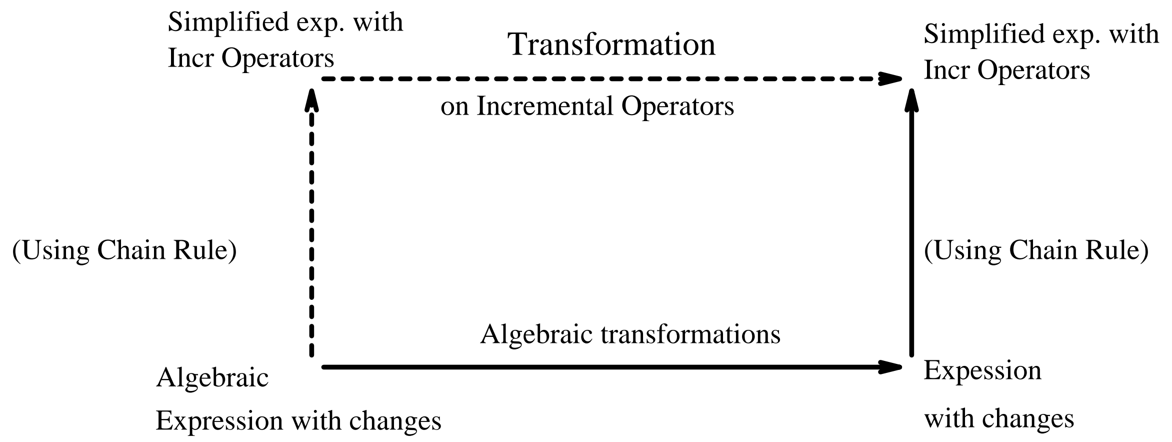

Figure 5.1. Two approaches for optimization.

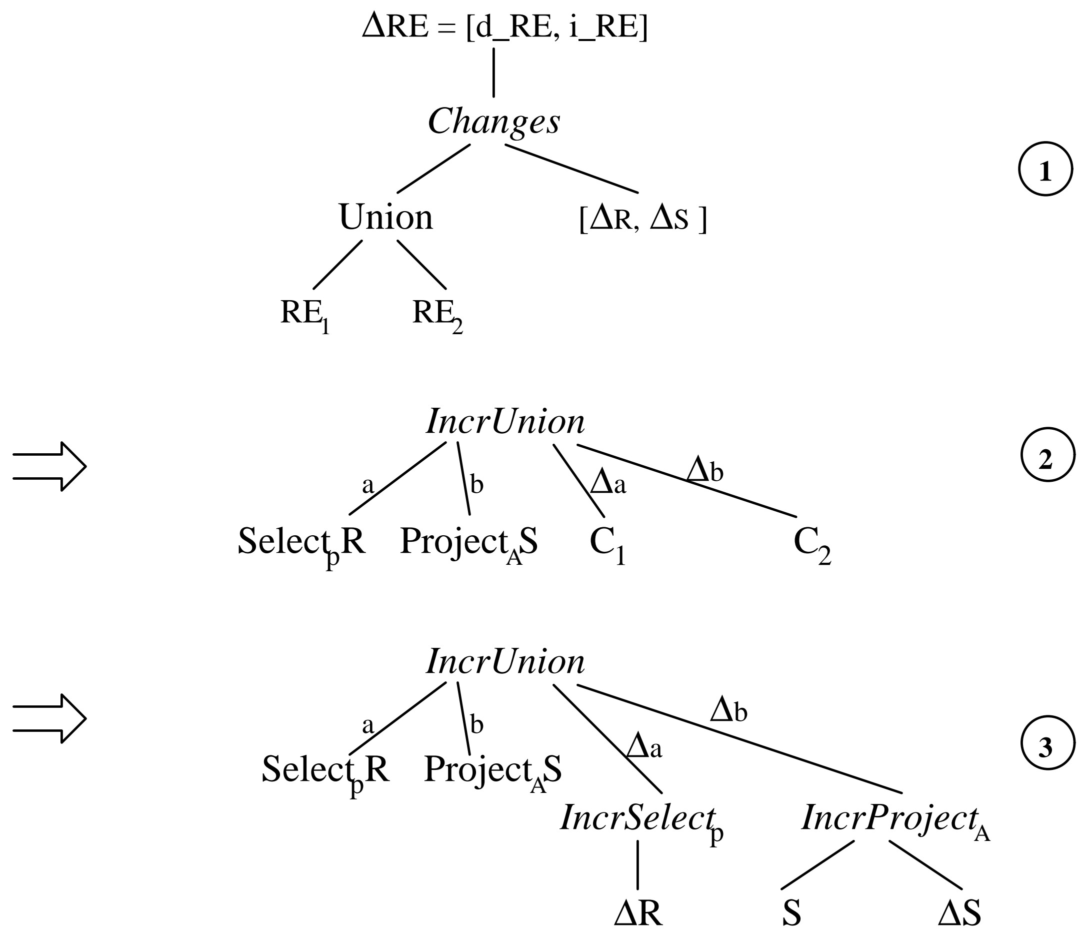

Figure 5.2. Illustration of the Generic Chain Rule.

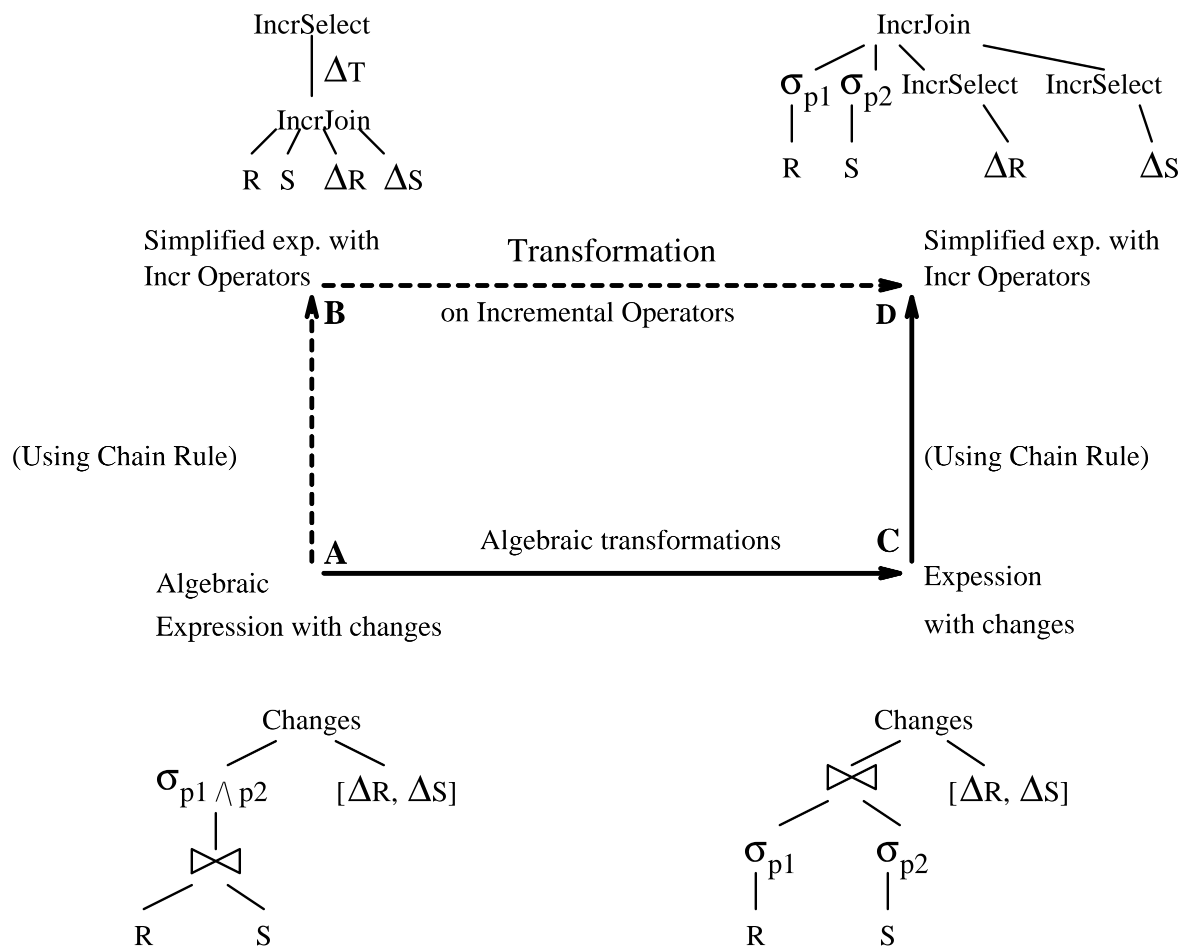

Figure 5.3. Illustration of the two Optimization Approaches.

CHAPTER 6
CONDITION EVALUATION IN DISTRIBUTED ENVIRONMENT

In a distributed environment, in addition to distributing different relations over different sites, it is common to divide the relations into sub-relations, called *fragments*, which are then allocated to a set of sites distributed over a network. The basic purpose of distribution is to allow locality and concurrency of access. There are two basic alternatives to fragmenting a base relation: *horizontal* and *vertical*. In *horizontal* fragmentation, a relation is partitioned along its tuples, that is, each fragment is an exclusive subset of the base relation. In *vertical* fragmentation, a relation is divided along columns. Each fragment contains a subset of attributes as well as the primary key of the base relation. In addition to these two fragmentation schemes, there exists a *hybrid* (or, mixed) scheme which is a *nested* combination of the first two. By *nested* we mean that a vertical fragmentation can be followed by a horizontal one, or vice versa, thus producing a tree like structure.

As we know that in the horizontal fragmentation case, the original relation can be obtained from the fragments by using the *Union* operator, *IncrUnion* is required in order to optimize conditions over horizontally distributed relations. Similarly, in the case of vertically fragmented relations we need *IncrJoin*. In the hybrid fragmentation case, conditions can be optimized by using both IncrUnion and IncrJoin, and the chain rule in conjunction with the transformations on incremental operators.

Before going into details of optimizing conditions in distributed databases, we need to address some issues in the design of distributed active databases. This is necessary because unless we know how the ECA rules look like in a distributed systems it is difficult to address optimization of condition. For instance, fragmentation of deltas

need to be considered as the base relations are already assumed to be fragmented in the above mentioned fashions.

6.1 Distributed Active Database Issues

Incorporating ECA rules into a distributed database raises several issues. Some of them are discussed in the following discussion.

Database Design: Data and processing needs of database *users* is considered as a general criteria for distributing base relations. However, this criteria is not sufficient in the presence of ECA rules in a distributed active database. Rule processing requirements also need to be given consideration and included in the criteria, which means that rules should also be as considered as *users* (or subscribers) in the above criteria. This is important because there may be cases when distribution of relations based on user queries is not good for rule evaluation. If we look at the structure of ECA rules, we find that condition part can be considered as a predefined query, however involving deltas also, in addition to base relations. Similarly, event (database event, to be specific) and action parts also access base relations.

We know that in the case of queries, locality of access is defined with respect to the site where the query is submitted. We keep the same criteria for rules also, that is, locality of access should be determined with respect to the site where the rule is submitted.

ECA-Rules: Most of the active systems allow rules to be defined over the base relations. However, as mentioned earlier it is important to have rules defined over views (or derived data). This is further more important in the distributed case as here a user typically works on a local schema which has its relations derived from the relations defined over the global schema and distributed over a network.

In distributed case, ECA rules can be classified into two categories: Inter-Site rules and Intra-Site rules. Rules in the former category accesses relations/deltas from multiple sites and the latter from only one site. The second type of rules do not involve communication overheads so are cheaper to evaluate.

The condition part of an ECA rule is a query which is automatically evaluated after the corresponding event become true. To put it differently, a condition subscribes to certain event (specified in the event part) from the event detector (see [17] for details) which informs the condition evaluator about the event and passes the needed parameters. When the condition becomes true (that is, returns a non-empty result) action is prompted for execution. Action is a transaction.

So, condition can be assumed as a pre-defined query which means that parts of it can be sent to other sites for efficient evaluation. However, there is a possibility of permanently storing those parts of conditions on those sites — as conditions remain unchanged over a time interval and require repeated evaluation. In addition as proposed by Chakravarthy et al. [11], multiple query optimization techniques will be useful to optimize a group of conditions evaluated simultaneously over a site. As the action part is a transaction, although a prestored one, it can be optimized in the similar way.

Delta Fragments: Deltas are different from base relations in certain ways. First, they are temporary. Second, they are small in size since a transaction normally operates on only a small subset of a base relation. Third, deltas are automatically fragmented. This is because a base relation fragment is usually not moved to a different site to perform an update transaction on it, which subsequently means that a corresponding delta fragment will be generated at the same site where the base relation fragment is allocated.

A delta captures the effect of a transaction on a base relation in the form of deleted, inserted and updated tuples. So they are needed only until rule evaluation is finished, or until commit (in most cases[1]) of the triggering transaction.

A delta can always be derived from the cache or the physical log of the triggering transaction depending on the recovery algorithm as it is the effect of a transaction. So, it does not require any extra overheads of disk access as a transaction has already fetched it. Besides, we can store delta in the main memory. This is because even in the case of system failure we need not worry about losing it since the triggering transaction (and also rule transaction(s)) will be undone during restart, after the system comes up.

As described above, the delta fragment corresponding to a base relation fragment is generated at the site where the the base relation is allocated. It is logical also to store it this way because, first, a delta can be considered as a part of the corresponding base relation (as it stores deleted, inserted and updated tuples of the corresponding relation), and second, all the incremental operators (except IncrSelect) need both the base relation and its corresponding delta as operands. This implies that whatever allocation is good for base relation fragments, the same is good for storing the corresponding deltas also. If the base relation fragments are replicated, there is no harm in replicating the corresponding delta fragments also, because we shall anyway have to update (with help of delta) each copy of the replicated base relation fragments once an update takes place on one of them.

In the remaining part of this section, we discuss condition evaluation when base relations are fragmented in horizontal, vertical and hybrid fashions.

[1]HiPAC [11] has proposed three basic coupling modes: immediate, deferred and decoupled, between the triggering and the triggered transactions. In all the coupling modes, except *nested top transaction* (a special kind of decoupled mode which we shall not consider in the present discussion), the whole rule processing activity has to end before the commit of the triggering transaction. For details, see the reference.

6.2 Horizontal Fragmentation

A horizontally fragmented relation can be written as an algebraic expression involving *Union* and the original relation can be recovered by just taking the union of the fragment relation. But, the same procedure is not valid for recovering the base delta from the *delta fragments*. We call them *delta fragments* just because they appear to be as if the delta of base relation is fragmented into delta fragments. The following lemma shows how to compute delta of a base relation, by using the delta fragments and relation fragments.

Lemma 1: If a relation R is horizontally partitioned into R_1 and R_2, and their respective deltas are $\triangle R_1$ and $\triangle R_2$ then delta of R is equal to the incremental union of R_1 and R_2. That is,

$$\triangle R \equiv IncrUnion(R_1, R_2, \triangle R_1, \triangle R_2) \tag{6.1}$$

Proof:

$$\begin{aligned} R &\equiv R_1 \cup R_2 \\ \triangle R &\equiv Changes(R_1 \cup R_2, [\triangle R_1, \triangle R_2]) \\ &\equiv IncrUnion(R_1, R_2, \triangle R_1, \triangle R_2) \end{aligned}$$

6.3 Vertical Fragmentation

In the case of vertical fragmentation, a base relation is divided into two or more relations, each fragment having a proper subset of the attributes of the base relation and including the key attributes. The base relation can be recovered by taking a natural join of the vertical fragments. Assuming that all the fragments of a base relation generate their own independent deltas, the following lemma evaluates delta of the base relation from the *delta fragments*.

Lemma 2: If a relation R is vertically fragmented into relations R_1 and R_2, and their respective delta fragments are $\triangle R_1$ and $\triangle R_2$ then delta of R is equal to the incremental join of R_1 and R_2. That is,

$$\triangle R \equiv IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2) \tag{6.2}$$

Proof:

$$\begin{aligned} R &\equiv R_1 \bowtie R_2 \\ \triangle R &\equiv Changes(R_1 \bowtie R_2, [\triangle R_1, \triangle R_2]) \\ &\equiv IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2) \end{aligned}$$

6.4 Hybrid Fragmentation

When a relation (say, R) is partitioned both horizontally and vertically, the original relation can be reconstructed by using Union and Join operators. Delta for R ($\triangle R$) can be computed by using the above lemmas. This is illustrated in the following example.

Example: Say a relation R is first fragmented into two vertical fragments R_1 and R_2 and then R_1 is horizontally fragmented into R_{11} R_{12}, and R_2 into two vertical fragments R_{21} and R_{22}. Then, using the above lemmas, we get:

$$\begin{aligned} \triangle R &\equiv IncrJoin(R_1, R_2, \triangle R_1, \triangle R_2) \\ &\equiv IncrJoin(R_{11} \cup R_{12}, R_{21} \bowtie R_{22}, IncrUnion(R_{11}, R_{12}, \triangle R_{11}, \triangle R_{12}), \\ &\qquad IncrJoin(R_{21}, R_{22}, \triangle R_{21}, \triangle R_{22})) \end{aligned}$$

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

Importance of evaluating changes to stored data and as well as to derived data has been recognized in the Active Database literature. But, however, extant systems keep their scope limited to stored relations only. In relational context changes to stored data means changes at attribute level (ie, updating an attribute of a tuple) and at relation level (removal and insertion of tuples from a relation). Changes to derived data means the same but here the relation is derived from others, it no more is just a stored relation.

Our algebra, ERA, provides a mathematical basis for evaluating changes to derived data. Our approach not only allows evaluation of changes to arbitrary non-aggregate relational expressions, but also facilitates in determining the optimized way of evaluating changes — in nearly the same way as optimizing relational expressions.

In this thesis, we have presented our algebra, ERA, which includes incremental operators for Select, Project, Join, Union and Set Difference. We have proposed a general version of IncrProject which is capable of handling duplicates in the projected relation. The definition of our IncrJoin eliminates problems present in the earlier work. Definitions of IncrUnion and IncrDiff are proposed and significance of IncrUnion and IncrJoin in the distributed case is shown. Our Chain Rule, for evaluating changes to complex relational expressions, can handle n-ary operators. Two approaches to optimizing conditions are proposed. We have also proposed optimizing transformations over incremental operators. Several issues involved in distributed

active databases are addressed. Proofs of all the operators and transformations are given.

There are several possibilities for future work in this direction. A few of those that we are currently working on are as follows:

- Develop incremental versions of aggregate operators and modify the chain rule to allow computation of changes to condition expressions involving aggregate operators.

- Extend the notion of delta and incremental operators to the object-oriented case, and implement them on Zeitgeist Object System. There is already some work on the specification of ECA rules in an object-oriented environment [32, 25].

- Distributed database design in the presence of triggers/rules. It has been proposed in this thesis that rules should be included in the criteria for distribution of relations.

- Using the algebra developed here for automatic generation of rules for view maintenance. Ceri and Widom [9] have proposed a procedural approach to this problem. We believe that our algebra shall not only provide a theoratical basis for incrememtal view maintenance but ECA rules will also allow 100% automation of rule generation and optimization for incremental update of views. Which means that rules can be directly derivable from the view definition and also optimized, without user intervention or help.

- Using multiple query optimization techniques for optimizing groups of condition expressions. These techniques need to be compared with the pattern matching algorithms used is AI, e.g., RETE and TREAT.

- Physical level optimization of incremental operators.
- Developing a single optimizer for conventional queries and ECA rules.
- To determine how the granularity of consistency enforcement relate to the units of concurrency, serializability and recovery etc. in the cases of nested, multilevel and long transactions, and in the presence of different coupling modes proposed in HiPAC [11].

APPENDIX
A COMPARISON WITH HiPAC APPROACH

In this appendix, we discuss HiPAC's definition of delta, Incremental Project, Incremental Join and Chain Rule.

Delta Relation and Schema Conversions   According to HiPAC's definition, the delta relation of the Figure 4.1, would look like as in the Figure A.1.

The pre-tilde attributes represent old value and post-tilde attributes represent new values. The delta is defined on a new schema, and deletions (including old values of the updated tuples) and insertions (including new values of the updated tuples) are outerjoined into a single relation. The effect of this is to put old and new values of the updated tuples into the same row, and to get a single result as an output of an incremental operator.

This definition of delta requires handling of different schemas and schema conversions (*pretilde* and *postilde* functions have been provided for this purpose). Special functions (*Removals*, *Additions*, *Removals*∼ and *Additions*∼) are needed to extract removals and insertions on different schemas. This definition of delta also necessitates an outerjoin on the output of every incremental operator, even though separate removals and insertions are usually computed.

HiPAC's Operators and Chain Rule   Our *Changes* and *IncrSelect* operators are similar to the ones proposed in HiPAC, however the latter requires an extra outerjoin and some special functions discussed above. Other operators are discussed as follows:

IncrProject: The following is the definition of HiPAC's IncrProject ([49]):

Let $A = \{A_1, \ldots, A_n\}$ denote a subset of the attributes in R, where $tid \in A$ and the predicate *differ* state that for some attribute A_i in A, $t. \sim A$ differes from $t.A\sim$. Then,

$$IncrProject_A(\triangle R) \quad = \quad Select_{differ}[Project_{\sim A \cup A \sim}(\triangle R)]$$

If we apply this definition to the delta in Example 2. The $\triangle V$ that we get is:

$\triangle V$	$\sim V_tid$	$\sim V_value$	$V_tid\sim$	$V_value\sim$
	01	799	01	400
	02	388	02	700
	04	999	–	–
	–	–	05	388
	–	–	06	400

One interesting observation in this delta is that TID of the tuple with value 388 is effectively modified (the tuples with IDs 02 and 05). This definition works well if the tuples having different TIDs but the same value are considered to be different entities. Or, in other words, this definition does not work if there are duplicates in the projected view.

If we look closely at the the new view (ie, V') generated by using the above $\triangle V$, we find that by removing duplicates, we shall get the correct new view. However, the same is not true for evaluating correct changes to the view by removing duplicates from the above $\triangle V$. Actually, it can be easily proved that the correct $\triangle V$ cannot be evaluated without using the base relation (ie, R) in some way or another. This is because we need the information about the number of duplicates present in the base relation.

Although it is not specified explicitly in this definition, it seems that A (set of projected attributes) must include all the key attributes (beside the TID column),

in which case the above mentioned problems will not arise because there will be no duplicates in the projected view.

<u>IncrJoin:</u> The following is the definition of IncrJoin in HiPAC:

$$IncrJoin([R_1, R_2, \triangle R_1, \triangle R_2], join_pred)$$

$$= LeftIncrJoin(\triangle R_1, R_2, join_pred) \cup$$

$$RightIncrJoin(R_1, \triangle R_2, join_pred) \cup$$

$$Both(\triangle R_1, \triangle R_2, join_pred)$$

Where,

$$LeftIncrJoin(\triangle R_1, R_2, join_pred)$$

$$= Outerjoin(Pretilde[Join(Removals(\triangle R_1), R_2, join_pred)],$$

$$Postilde[Join(Additions(\triangle R_1), R_2, join_pred)]$$

$$(\sim tid_1 = tid_1 \sim) \wedge (\sim tid_2 = tid_2 \sim))$$

$$RightIncrJoin(R_1, \triangle R_2, join_pred)$$

$$= Outerjoin(Pretilde[Join(R_1, Removals(\triangle R_2), join_pred)],$$

$$Postilde[Join(R_1, Additions(\triangle R_2), join_pred)],$$

$$(\sim tid_1 = tid_1 \sim) \wedge (\sim tid_2 = tid_2 \sim))$$

$$Both(\triangle R_1, \triangle R_2, join_pred)$$

$$= Outerjoin(Pretilde[Join(Removals(\triangle R_1), Removals(\triangle R_2), join_pred)],$$

$$Postilde[Join(Additions(\triangle R_1), Additions(\triangle R_2), join_pred)],$$

$$(\sim tid_1 = tid_1 \sim) \wedge (\sim tid_2 = tid_2 \sim))$$

This definition of IncrJoin does not give the correct results as can be seen if we apply it to the relations of Example 3:

$\triangle V$	$\sim R_1_tid, \sim S_2_tid$	$\sim R_1_val$	$\sim S_2_val$	$R_1_tid\sim, S_2_tid\sim$	$R_1_val\sim$	$S_2_val\sim$
	03,20	179	179	03,20	700	179
	03,20	179	179	03,20	–	–
	03,20	179	179	03,20	–	–
	03,30	179	200	03,30	700	200
	03,30	179	200	03,30	179	645
	03,30	179	200	03,30	700	645

This is a partial result of what we shall get by taking a Union of Left, Right and Both IncrJoins. This shows that one single tuple (eg, $03, 30$) has been modified in three different ways which obviously is incorrect. Besides, a lot of duplicates are generated. One thing that has been observed is that the above definition works fine when one of the deltas is null, or when there are no deletions (ie, $R = R - d_R$).

Chain Rule   The following is HiPAC's Chain Rule:

$$
\begin{aligned}
& Changes(F \circ G; [RList, \triangle RList]) \\
= \quad & IncrF(G(RList), Changes(G; [RList, \triangle RList]))
\end{aligned}
$$

In this expression, RList is a list of relation and $\triangle RList$ is the list of corresponding deltas; F and G can be any SPJ operators. IncrF is the *Incremental* form of the operator F.

This definition of chain rule can handle only unary operators, e.g., Select and Project. Our chain rule can handle any k-ary operators.

Transaction T: Modify(@111, new_tuple = (@111, Mike, 45K));
Insert(@444, Sharon, 30K);
Delete(@333)

(a)

EMP	TID	Name	Salary
	@111	Mike	40K
	@222	Carol	35K
	@333	John	30K

(b)

EMP'	TID	Name	Salary
	@111	Mike	45K
	@222	Carol	35K
	@444	Sharon	30K

(c)

$\triangle$ EMP	$\sim TID$	$\sim Name$	$\sim Salary$	$TID\sim$	$Name\sim$	$Salary\sim$
	@111	Mike	40K	@111	Mike	45K
	@333	John	30K	-	-	-
	-	-	-	@444	Sharon	30K

(d)

Figure A.1. Structure of HiPAC's delta. (a) A transaction, T, on EMP. (b) A relation, EMP. (c) EMP after the transaction. (d) Delta relation of EMP.

REFERENCES

[1] J. Albert. Algebraic Properties of Bag Data Types. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 211–219, Barcelona, Spain, Sep. 1991.

[2] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 16–52, Washington DC, May 1986.

[3] F. Barbic and B. Pernici. Time Modeling in Office Information Systems. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 51–62, Los Angeles, May 1985.

[4] J. Blakely, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 61–71, Washington DC, May 1986.

[5] D. Bobrow and T. Winograd. An Overview of KRL, A Knowledge Representation Language. *Cognitive Science*, 1(1):3–46, 1977.

[6] D. G. Bobrow and M. Stefik. *The Loops Manual*. Intelligent Systems Laboratory, Xerox Corporation, Boston, MA, 1983.

[7] P. Buneman and E. Clemens. Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems*, 4(3):368–382, Sep. 1979.

[8] M. Carey, D. DeWitt, and S. Vandenberg. A Data Model and Query Language for EXODUS. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 29-36, Chicago, Illinois, May 1988.

[9] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, Sep. 1991.

[10] S. Chakravarthy. Divide and Conquer: A Basis for Augmenting a Conventional Query Optimizer with Multiple Query Processing Capabilities. In *Proc. of the 7th Int'l Conf. on Data Engineering*, pages 482–490, Kobe, Japan, Apr. 1991.

[11] S. Chakravarthy, B. Blaustein, A. P. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhary, R. Ladin, M. Livny, D. McCarthy, R. McKee and A. Rosenthal. HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, Aug. 1989.

[12] S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kaufmann Publications, Boston, MA, 1988.

[13] U. S. Chakravarthy. Rule management and Evaluation: An Active DBMS Perspective. *Special issue of ACM SIGMOD Record on rule processing in databases*, 18(3):20–28, Sep. 1989.

[14] U. S. Chakravarthy and J. Minker. Multiple Query Processing in Deductive Databases Using Query Graphs. In *Proceedings of International Conference of Very Large Data Bases*, pages 384–391, Houston, TX, 1986.

[15] S. Chakravathy and R. Blanco-Mora. Supporting very large production systems using active dbms abstraction. Technical Report UF-CIS TR-91-25, Database Systems R&D Center, University of Florida, Gainesville, Sep. 1991.

[16] S. Chakravathy and S. Garg. Extended relational algebra (era): for optimizing situations in active databases. Technical Report UF-CIS TR-91-24, Database Systems R&D Center, University of Florida, Gainesville, Nov. 1991.

[17] S. Chakravathy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, Database Systems R&D Center, University of Florida, Gainesville, Sep. 1991.

[18] S. Chakravathy, S. B. Navathe, S. Garg, D. Mishra, and A. Sharma. An evaluation of active dbms developments. Technical Report UF-CIS TR-90-23, Database Systems R&D Center, University of Florida, Gainesville, Sep. 1990.

[19] Hawru S. Chuang. Operational Rule Processing in a Prototype OSAM* Knowledge Base Management System. Master's thesis, University of Florida, Database Systems R&D Center, Gainesville, Aug. 1990.

[20] K. L. Clark. Negation as failure. In H. Galliare and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press Publishing Co., New York, 1978.

[21] M. Darnovsky and J. Bowman. *TRANSACT-SQL USER'S GUIDE*. Document 3231-2.1, Sybase Inc., New York, 1987.

[22] U. Dayal, A. Buchmann, and D. McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database Management System. In *Proceedings 2nd International Workshop on Object-Oriented Database Systems*, pages 100–114, Ebernburg, West Germany, Sep. 1988.

[23] U. Dayal. The HiPAC project: Combining active databases and timing constraints. *Special Issue of Real Time Data Base Systems, SIGMOD Record*, 17(1):51–70, Mar. 1988.

[24] U. Dayal, N. Goodman, and R. H. Katz. An Extended Relational Algebra with Control over Duplicate Elimination. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 79–87, Los Angeles, CA, Mar. 1982.

[25] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 120–127, Barcelona, Spain, Sep. 1991.

[26] K. R. Dittrich, A. M. Kotz, and J. A. Mulle. An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases. *SIGMOD Record*, 15(3):22–36, Sep. 1986.

[27] K. P. Eswaran. Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated Data Base System. IBM Research Report RJ1820, San Jose, CA, Aug. 1976.

[28] K. P. Eswaran and D. D. Chamberlain. Functional Specifications of a Subsystem for Data Base Integrity. In *Proceedings 1st International Conference on Very Large Data Bases*, pages 213-219, Los Angeles, CA, Sep. 1975.

[29] S. Finkelstein. Common Expression Analysis in Database applications. In *Proceedings ACM-SIGMOD*, pages 79-84, Orlando, FL, Jun. 1982.

[30] C. L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem. *Artificial Intelligence*, 19(2):17–37, 1982.

[31] C. L. Forgy and J. McDermott. Domain-Independent Production System Language. In *Proceedings Fifth International Conference on Artificial Intelligence*, pages 121-129, Cambridge, MA, 1987.

[32] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelon, Spain, Sep. 1991.

[33] Eric N. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings 1987 ACM SIGMOD Conference on Management of Data*, pages 440–453, Miami, FL, May 1987.

[34] Eric N. Hanson. An Initial Report on the Design of Ariel: a DBMS with an integrated production rule system. *ACM SIGMOD RECORD*, 18(3):12–19, Sep. 1989.

[35] Eric N. Hanson, Moez Chaabouni, Chang ho Kim, and Yu wang Wang. A predicate matching algorithm for database rule systems. In *Proceedings 1990 ACM SIGMOD International Conference on management of data*, pages 64-71, Atlantic City, NJ, May 1990.

[36] Eric N. Hanson and Jennifer Widom. Active database rule systems. Personal communication, Wright State University, Mar. 1991.

[37] S. Hudson and R. King. CACTIS: A Database System for Specifying Functionally-Defined Data. In *Proceedings 1st International Workshop on Object-Oriented Database Systems*, pages 26–37, Dallas, TX, Sept. 1986.

[38] InterBase Software Corporation. *InterBase DDL Reference Manual, InterBase Version 3.0*, Bedford, MA, 1990.

[39] S. Koenig and R. Paige. A Transformational Framework for the Automatic Control of Derived Data. In *Proceedings 7th International Conference on Very Large Data Bases*, pages 306–318, Sept. 1981.

[40] A. M. Kotz, K. R. Dittrich, and J. A. Mulle. Supporting Semantic Rules by a Generalized Event/Trigger Mechanism. In *Proceedings International Conference on Extended Data Base Technology*, pages 201-208, Venice, Mar. 1988.

[41] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A Snapshot Differential Refresh Algorithm. In *Proceedings 1986 ACM SIGMOD Conference on Management of Data*, pages 53–60, Washington DC, Jun. 1986.

[42] J. Minker. Performing inferences over relational databases. In *Proc. of ACM SIGMOD*, pages 79–91, San Jose, CA, May 1975.

[43] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of AAAI 87 Conference on Artificial Intelligence*, pages 42–47, Boston, MA, Aug. 1987.

[44] M. Morgenstern. Active Databases as a Paradigm for Enhanced Computing Environments. In *Proceedings 9th International Conference on Very Large Data Bases*, pages 34–42, Los Angeles, CA, Sep. 1983.

[45] J. Moss. Nested Transactions: An Approach To Reliable Distributed Computing. MIT Laboratory for Computer Science, MIT/LCS/TR-260, 1981.

[46] John Mylopoulos and Michael L. Brodie. *Artificial Intelligence and Databases.* Morgan Kaufmann, New York, 1989.

[47] William T. Olle. *The CODASYL Approach to Data Base Management.* John Wiley & Sons, New York, 1978.

[48] L. Raschid and S. W. Su. Transaction-Oriented Mechanism to Control Processing in a Knowledge Base Management System. In *Proceedings 2nd International Conference on Expert Database Systems*, pages 163–174, Boston, MA, 1988.

[49] A. Rosenthal, U. S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation Monitoring in Active Databases. In *Proc. of the 15th Int'l Conf. on Very Large Databases*, pages 455–464, Amsterdam, Aug. 1989.

[50] N. Roussopoulos. View Indexing in Relational Databases. *ACM Transactions on Database Systems*, 7(2):258–290, Jun. 1989.

[51] P. Schwartz. Extensibility in the Starburst Database Systems. In *Proceedings Int'l Workshop on Object-Oriented Database Systems*, pages 223-230, Pacific Grove, CA, 1986.

[52] P. Selinger, M. Astraham, D. Chamberlin, R. Lorie and T. Price. Access path selection in a relational database management system. In *Proceedings 1979 ACM SIGMOD International Conference on Management of Data*, pages 178-184, Chicago, IL, Jun. 1979.

[53] T. Sellis, C. Lin, and L. Raschid. Data intensive production systems: the DIPS approach. *SIGMOD Record*, 18(3):102-114, Sep. 1989.

[54] T. Sellis and N. Roussopoulos. Deep Compilation of Large Rule Bases. In *Proceedings 2nd International Conference on Expert Database Systems*, pages 46–53, Baltimore, MD, Apr. 1988.

[55] Madhulika Singh. Transaction Oriented Rule Processing in An Object-Oriented Knowledge Base Management System. Master's thesis, University of Florida, Database Systems R&D Center, Gainesville, Aug. 1990.

[56] M. Stonebraker. On Rules, Procedures, Cashing and Views in Database Systems. In *Proceedings of ACM-SIGMOD*, pages 281–290, Atlantic City, NJ, May 1990.

[57] M. Stonebraker, E. Hanson, and C.H. Hong. The Design of the POSTGRES Rules System. The POSTGRES Papers (M. Stonebraker and L.A. Rowe, eds.), Memorandum No. UCB/ERL/ M86/85, Electronics Research Laboratory, University of California, Berkeley, CA, Nov. 1986.

[58] M. Stonebraker, M. Hanson, and S. Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, Jul. 1988.

[59] M. Stonebraker, M. Hearst, and S. Potaminos. A commentary on the POSTGRES Rule System. *SIGMOD Record*, 18(3):145–153, Sep. 1989.

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

[61] M. R. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, Sept 1976.

[62] A. Tanaka, S. B. Navathe, S. Chakravarthy, and S. Karlapalem. Er-r: An enhanced er model with situation-action rules to capture application semantics. In *Proceedings International Conference on the Entity Relationship Approach*, pages 59–75, Lausanne, Switzerland, Oct. 1991.

[63] J. Widom, R. J. Cochrane, and . Lindsay B, G. Implemented Set-Oriented Production Rules as an Extension of Starburst. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 275–286, Barcelona, Spain, Sep. 1991.

[64] J. Widom and S. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proc. of ACM-SIGMOD*, pages 259–270, Atlantic City, NJ, May 1990.

[65] Patrick Henry Winston and Richard Henry Brown. *Artificial Intelligence: An MIT Perspective.* The MIT Press, Cambridge, Massachusetts, 1979.

[66] M. Zloof. Office-by-example: a business language that unifies data and word processing and electronic mail. *IBM Systems Journal 21*, 4(3):272–304, 1982.

BIOGRAPHICAL SKETCH

Sudhanshu Garg was born on November 16, 1966 in Meerut, India. He received his Bachelor of Engineering degree in Computer Science & Technology from the University of Roorkee, India. After finishing his undergraduate degree in 1989, he worked as an Engineer at Larsen and Toubro Limited, Bombay. In the Spring of '90, he started his graduate studies with a major in Computer & Information Sciences at the University of Florida. He will receive his Master of Science degree in Computer and Information Sciences from the University of Florida, Gainesville in May, 1991.

His research interests include high performance databases, computer networks and expert systems. Specifically, he is interested in active, object-oriented and heterogeneous databases, and efficient implementations of network protocols.