STATIC ANALYSIS OF ECA RULES AND USE OF THESE RULES FOR INCREMENTAL COMPUTATION OF GENERAL AGGREGATE EXPRESSIONS

By

SEUNG-KYUM KIM

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1996

Dedicated to My Parents

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Sharma Chakravarthy for the continuous guidance and support during the course of this work. I thank Dr. Eric Hanson, Dr. Herman Lam, Dr. Stanley Su, and Dr. Suleyman Tufekci (in alphabetic order) for serving on my supervisory committee and for their perusal of this dissertation. I would like to thank Mrs. Sharon Grant for keeping the warm and comfortable research environment. I also thank many fellow students at the Database Systems R&D Center for their help and friendship.

Finally, I am deeply indebted to my parents for their endless sacrifice and love to me.

This work was supported in part by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vi

ABSTRACT vii

CHAPTERS 1

1 INTRODUCTION 1

1.1 Active Databases 1
1.2 Data Warehouses 4
1.3 Problem Statement 6
1.3.1 Support for Alternative User Requirements in Active Rules Execution 6
1.3.2 Efficient Support of Aggregates in Data Warehouses 8
1.3.3 Structure of the Dissertation 9

2 STATIC ANALYSIS OF ACTIVE RULES 10

2.1 Introduction 10
2.2 Limitations of the Earlier Rule Execution Models 13
2.3 Assumptions and Definitions 16
2.3.1 Rule Execution Sequence (RES) and Rule Commutativity 16
2.3.2 Dependencies and Dependency Graph 20
2.3.3 Trigger Graph 22
2.4 Confluence and Priority Specification 23

3 IMPLEMENTATION OF CONFLUENT RULE SCHEDULER 30

3.1 Strict Order-Preserving Rule Execution Model 30
3.1.1 Extended Execution Graph 30
3.1.2 Strict Order-Preserving Executions 32
3.1.3 Implementation 33
3.1.4 Parallel Rule Executions 36
3.2 Alternative Policies for Handling Overlapping Trigger Paths 40
3.2.1 Serial Trigger-Path Executions 40
3.2.2 Serializable Trigger-Path Executions 42
3.2.3 Comparisons with Strict Order-Preserving Execution 43
3.3 Discussion and Conclusions 45

4 AGGREGATE CACHE 48

4.1 Motivation 48
4.2 Updating Cached Aggregates 51
4.3 Incremental Update of Aggregates 56
4.3.1 Syntactic Conventions 56
4.3.2 Incrementally Updatable Aggregates 57
4.3.3 Algebraic Aggregates and Non-Algebraic Aggregates 58
4.3.4 Summative Aggregates 61
4.3.5 Binding of Variables 66
4.3.6 Decomposition of Summative Aggregates 68
4.3.7 Normalization of Summative Aggregates 73
4.3.8 Incremental-Updatability of Nested Summative Aggregates . . 79
4.4 Looking-Up Cached Aggregates 84
4.5 Conclusions 88

5 CONCLUSIONS 90

REFERENCES 92

BIOGRAPHICAL SKETCH 95

LIST OF FIGURES

2.1 Rule execution graphs 14
2.2 Overlapped trigger paths 15
2.3 A conflicting rule set 24
2.4 Priority graphs for Figure 1.4 25
2.5 A pair of trigger graph and dependency graph 27
2.6 A priority graph 27
2.7 An execution graph 28
3.1 Overlapping trigger paths and extended execution graph 31
3.2 Three different orderings of dependency edges in Figure 3.1(b) 32
3.3 Extended execution graph in strict order-preserving executions 32
3.4 Extended execution graph with rule_counts 34
3.5 Algorithm – Build_EG() 35
3.6 Algorithm – Schedule() 37
3.7 A priority graph with absolute priorities 41
3.8 Translation to serializable trigger-path execution – Step 1 43
3.9 Translation to serializable trigger-path execution – Step 2 44
4.1 A Data Warehouse and Aggregate Cache 50

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

STATIC ANALYSIS OF ECA RULES AND USE OF THESE RULES
FOR INCREMENTAL COMPUTATION OF
GENERAL AGGREGATE EXPRESSIONS

By

Seung-Kyum Kim

May, 1996

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Science and Engineering

In this work we address two major issues that are related within the framework of active databases. Firstly, we propose a practical approach to rule analysis. We show how alternative rule designer choices can be supported using our approach to achieve confluent rule execution in active databases. Our model employs priority information to resolve conflicts between rules, and uses a rule scheduler based on the topological sort to achieve correct confluent rule executions. Given a rule set, a trigger graph and a dependency graph are built from the information obtained by analyzing the rule set at compile time. The two graphs are combined to form a priority graph, on which the user is requested to specify priorities (or resolve conflicts) only if there exist dependencies in the dependency graph. The user can have multiple priority graphs by specifying different priorities depending on application semantics. From a priority graph, an execution graph is derived for *every user transaction* that triggers one or more rules. The rule scheduler uses the execution graph. Our model also correctly

handles the situation where trigger paths of rules triggered by a user transaction are overlapping, which are not handled by existing models. We prove that our model achieves maximum parallelism in rule executions.

Next, we propose a cache mechanism, called *aggregate cache* for efficiently supporting complex aggregate computations in data warehouses. We discuss several cache update strategies in the context of maintaining consistency between base databases and aggregates cached in the data warehouse. We formally define the incremental update of aggregates, which is a prime issue for the aggregate cache. Further we classify algebraic aggregates into *summative aggregates* that include a vast variety of aggregates applicable in data warehouses to support decision making and statistical data analysis. We prove that there is a precise subclass of summative aggregates that can be incrementally updated. For the incrementally updatable class of summative aggregates, we propose an efficient cache mechanism that allows many user-queries to share accesses to the cached aggregates in a transparent way.

CHAPTER 1
INTRODUCTION

1.1 Active Databases

For the past decade, making the traditional passive databases *active* by incorporating a set of rules has drawn a lot of attention from the database research and development community. Originated from the concept of *triggers* proposed for the System R [16] and largely developed from the production rule languages for expert systems such as OPS5 [9], the research on active databases is now getting matured and several active database systems are being (or were) implemented including HiPAC [12], Ode [19], Postgres [35], Starburst [39], Samos [17], Ariel [29], and Sentinel [13].

While there are variations, a representative active database paradigm is to use the ECA rules (Event-Condition-Action) [12] with either the set-oriented semantics or the tuple-oriented (instance-oriented) semantics over the relational or Object-Oriented database. An ECA rule consists of three parts, event, condition, and action parts. Events usually correspond to database operations, especially data manipulation operations such as insert, delete, and update. For some systems as HiPAC and Sentinel, temporal events and external events (e.g., user signals) are included too. All of these events are called primitive events. For the Object-Oriented model, a method call is regarded as an event as well. Conditions are generally assumed to be predicates over parameters and database queries without side effects. An action consists of a set of data manipulation operations or a function call.

When an event occurs in the system, rules whose event part corresponds to the event occurred are triggered. Of the triggered rules, one rule is picked based on some

criteria (or by a process known as conflict resolution). Then, the condition part of the selected rule is tested. If the condition is satisfied, the action part of the rule is executed. The process of picking one triggered rule, testing condition, and executing action is repeated until no more triggered rules remain.

For event specification and detection in active databases (adopting the ECA rule paradigm), three major approaches have been taken. The common goal in these approaches is to support composite events. A composite event is a composition of primitive events and/or other composite events. For instance, in Sentinel [13], given two events E_1 and E_2, a disjunction of them, $E_1 \nabla E_2$ is defined to occur when either E_1 or E_2 occurs. While similar sets of composite events are defined in all the approaches, distinctions are found in the ways of detecting such composite events. In Ode, finite automata are used to detect composite events expressed by a variation of regular expression [21, 20], while in Samos, a labeled Petri Net is adopted [18]. In Sentinel, on the other hand, we use an event graph where each leaf node represents a primitive event and an intermediate node represents a composite event consisting of other events represented by its child nodes. When a primitive event occurs, the occurrence is propagated to its parent node. A parent node, with an appropriate restriction for each type of composite event, collects occurrences of its child events, notifies an occurrence of the composite event if a certain condition is met, and propagates the occurrence of the composite event to its own parent node [13].

For condition specification and testing, there are few systems that take a sophisticated approach, except Ariel. In case a condition is represented by a database query, that query should not update database contents (i.e., side effect free), and it is interpreted as satisfied if the query returns a nonempty result. In Ariel, condition testing is done by an algorithm, called A-TREAT [28]. It uses the discrimination network to efficiently compare a large number of patterns to a large number of data

without repetitive scanning. A-TREAT can speed up rule processing in a database environment and reduce the storage requirement of TREAT algorithm.

On the other hand, there are two representative rule execution semantics, *tuple-oriented* (or *instance-oriented*) semantics and *set-oriented* semantics [30]. When an event occurs, it can triggers one or more rules. These triggered rules are called instances of their respective rules. Even for one rule, there may exist multiple instances of one rule for any period of time. It should be noted that as events, granularities of data manipulation operations are quite coarse. For instance, an event of insert to a relational table is usually defined to occur when any tuple is inserted into the table. In other words, the granularity of the insert event is the whole relation, not a tuple. Also, in SQL, an insert, delete, or update statement can modify multiple tuples in one execution. Therefore if a rule is triggered by any of the data manipulation operations, it is likely that such an event will give rise to multiple instances of the same rule. The two rule execution semantics make a difference in such situations. Suppose one execution of insert statement inserts three tuples, t_1, t_2, and t_3, into a table. And a rule r_1 is triggered by that insert event. In the tuple-oriented semantics, t_1, t_2, and t_3 trigger their own instance of r_1, and for each instance of r_1, its condition is tested, and if satisfied, its action is executed. However, in the set-oriented semantics, only one instance of r_1 is executed for t_1, t_2, and t_3; that is, r_1's condition is tested just *once* and if the condition is satisfied, the action part of r_1 is carried out on each of t_1, t_2, and t_3. Postgres has the tuple-oriented rule execution semantics, while Starburst and Ariel have the set-oriented semantics. HiPAC and Sentinel, on the other hand, have rather different a rule execution semantics. In these systems, conceptually all triggered rules are executed concurrently using the nested transaction model [12, 6].

1.2 Data Warehouses

In order to support efficient processing of queries which are often of complex forms and span over vast amount of data which in turn could be distributed and even heterogeneous, one viable approach is to maintain a separate dedicated repository that holds the gist of base data and to process the queries over the repository. By doing so, it is expected that decision support applications that usually involve lots of such otherwise expensive queries can also be detached from ordinary online transactions being performed over the underlying base databases, thereby significantly enhancing performance of both categories of applications. This approach, known as *data warehousing* [31], is rapidly becoming popular, and a lot of research effort is recently being put to solve various related issues [33].

A data warehouse system can be viewed as a multi-layered database system. At the bottom level, there are several databases, called *base databases*, each of which is an operational, independent database system. At the top level, there is one specialized, (almost) read-only database system that contains the most abstracted or summarized data derived from the databases in one level below. As a result, this system constitutes a pyramid-structured abstraction of information (or data). But usually only a two-layer structure is assumed; that is, base databases and a summarized database which we call *data warehouse*. Between the two layers, a physical separation is generally assumed. The base databases can also be distributed and even heterogeneous. All these separated databases are connected through a network.

Since the data warehouse and base databases are separated and the data warehouse contains information derived from the base databases, it is a natural requirement that as base databases change, the data warehouse should be updated accordingly, if the changes are relevant to the information in the data warehouse. Therefore,

an intermediate layer between the two layer would be necessary to mediate communications between the two layers. This intermediate layer is responsible for monitoring changes made to the base databases and propagating them to the data warehouse to update its contents. In practice, however, the intermediate layer can be two interface layers, each of which resides in each base database and in the data warehouse. The interface layer in base databases is responsible for detecting changes in its respective base database and it should communicate with the interface layer in the data warehouse to propagate the changes. One issue arising here is when to or how often to propagate the changes. Proposed approaches in the literature include *eager propagation* for a currency critical data set, *polling* for a data set that changes slowly or whose currency requirement is not critical, and *lazy (or on-demand) propagation* for a data set that changes slowly and is not used frequently.

A closely related issue to the change propagation is the *incremental update* of the data warehouse. It is unthinkable to repopulate (or re-derive) the data warehouse whenever a change is made in a base database. Therefore, there should be a way to incrementally update the data warehouse with the propagated changes. In fact, this issue is not a new one. View materialization [8, 27] in the relational database deals with a similar problem, but not identical. Although the data warehouse can be viewed as a set of materialized views (derived from base databases), unlike view materialization, the data warehouse has a lot of problems that need special attention. The views defined in the data warehouse can be very complex for which the conventional view materialization techniques are inadequate. They usually contain historical data and highly aggregated and summarized data. This is another reason why the view materialization techniques cannot be directly applied [37]. Recently, the issue of view materialization has been revisited by several researchers including [23, 24].

1.3 Problem Statement

1.3.1 Support for Alternative User Requirements in Active Rules Execution

Using ECA rules in active database systems for real-life applications involves implementing, debugging, and maintaining a large number of rules. Experience in developing large production rule systems has amply demonstrated the need for understanding the behavior of rules especially when their execution is non-deterministic. Availability of rules in active database systems and their semantics create additional complexity for both modeling and verifying the correctness of such systems.

For the effective deployment of active database systems, there is a clear need for providing an analysis (both static and dynamic) and explanation facility to understand the interaction–among rules, between rules and events, and between rules and database objects. Due to the event-based nature of active database systems, special attention has to be paid for making the context of rule execution explicit [15]. Unlike an imperative programming environment, rules are triggered in the context of the transaction execution and hence both the order and the rules triggered vary from one transaction/application to another.[1]

Ideally, the support environment needs to be able to accept expected behavior and compare it with the actual behavior to provide a useful feedback on the differences between the two. This has to be done in the context of database objects, rules, events, and transaction sets that are executed concurrently.

Short of this, it is useful to provide alternatives that allow the designer to choose among various options that are meaningful to his/her application. For example, user

[1] Use of the nested transaction model for rule execution in Sentinel [6, 36] provides such a context. Our graphics visualization tool [14] using Motif displays transaction and rule execution by using directed graphs to indicate both the context (i.e., transactions/composite events) and the execution of (cascading) rules. We plan to augment the existing visualization capability with the static analysis proposed in this paper.

requirements may come from the following set of answers as far as rule execution is concerned:

1. No preference; any arbitrary execution order of rules and their final result is acceptable.

2. Arbitrary final state is NOT acceptable. The designer wants to see a unique final result whenever the same set of rules executes from the same initial database state (i.e., Confluent Rule Execution is desirable).

3. This particular group of rules must give a unique result when any subset of these rules executes. No preference for the rest of the rules.

4. This application must have this order and that application must have that order, if a different execution order gives a different result.

It is evident that the ability to interact with the designer and to support alternative user requirements is quite important. As a result, support environments for the design of ECA rules in the context of active databases need to support:

- ECA rule analysis to provide feedback on the interaction of rules either globally or with respect to transactions,

- Confluence analysis at rule scheduling time (in addition to static analysis),

- Parallel rule execution where possible for efficiency reasons, and

- Visualization of actual run time rule execution and related contextual information.

In the first part of this thesis (Chapters 2 and 3), we propose an approach that addresses the above. We address confluent rule executions (which deal with obtaining a unique final database state for any initial set of rules) for a user transaction. We

show that previous rule execution models are inadequate for confluent rule executions in some cases, and propose extensions that can readily meet the alternative user requirements with respect to confluent executions. We also show that our model naturally supports parallel rule executions.

1.3.2 Efficient Support of Aggregates in Data Warehouses

Although for years there has been great interest in the data warehouse within the database industry and within academia as well, it does not appear that an efficient, fully functional, and flexible data warehouse system has emerged yet. There are many reasons for that. Part of them are purely engineering problems. There are a lot of legacy database systems that are still running and lack interface capability with any other new systems. Extracting data from such a system and monitoring data changes may not be very theoretically challenging, but from the engineering point of view, they are not trivial tasks at all. On the other hand, there are still issues to be answered in more systematic and fundamental ways. One such issue that we find very important is the efficient support of aggregates in data warehouses. As data warehouses contain a lot of highly summarized data and many applications in data warehouses perform from simple to very complex analyses over the data, it is crucial to improve performance of such aggregate or summary operations. In a naive implementation, it could take more than several hours to perform a simple summation over millions of records dispersed in base databases. With such a response time, no one would expect an interactive data analysis, which is an important requirement for the data warehouse to be indeed useful as a cooperative tool.

With this consideration in mind, in Chapter 4, we propose a practical means to boost the performance of aggregate processes in data warehouses. The aggregate cache that we propose saves in the system (i.e., the data warehouse) previous results of aggregate computations. As base databases change, the stored results are updated

appropriately. Along with such an engineering aspect, in our work we identify and prove that there is an exact class of aggregates that are guaranteed to be incrementally updatable, a result that we believe is a theoretical contribution to the research on data warehouse.

1.3.3 Structure of the Dissertation

The rest of this dissertation is structured as follows. In Chapter 2, we introduce formal definitions of confluent rule execution and show conditions in which confluent rule execution can be achieved. In Chapter 3, we generalize the notions developed in Chapter 2 to obtain confluent rule execution in general situations and rule scheduling algorithms along with a proof of maximal parallelism that our approach attains. In Chapter 4, we propose the aggregates cache and identify and prove a class of aggregates that can be incrementally updatable. Chapter 5 contains general conclusions for this dissertation.

CHAPTER 2
STATIC ANALYSIS OF ACTIVE RULES

2.1 Introduction

Incorporating ECA rules (Event-Condition-Action rules) into the traditional database systems, thereby making them active databases, can broaden database applications significantly [5, 12, 19, 30]. Also, ECA rules provide more flexible and general alternatives for implementing many database features, such as integrity constraint enforcement, that are traditionally hard-wired into a DBMS [10, 11, 35, 38, 39].

An ECA rule consists of three parts: *event*, *condition*, and *action* parts. Execution of ECA rules goes through three phases: event detection, condition test, and execution of action. An event can be a data manipulation or retrieval operation, a method invocation in Object-Oriented databases, a signal from timer or the users, or a combination thereof. An active database system monitors occurrences of events pre-specified by ECA rules. Once specified events have occurred, the condition part of the relevant rule is tested. If the test is satisfied, the rule's action part can be executed. In Sentinel [13], a rule is said to be *triggered* when the rule has passed the event detection phase; that is, when one or more events which the rule is waiting for have occurred. When an ECA rule has passed the condition test phase, it is said to be *eligible* for execution.[1] In this work, we use "trigger" to describe the eligible rules assuming that the condition part has been satisfied or it is nil.

[1]The definition of *trigger* is blurred as condition-action rules such as the production rule [9] have evolved to ECA rules. A condition-action rule is triggered and eligible for execution when the current database state satisfies the specified condition, whereas for an ECA rule to be ready to execute, it has to pass separate event detection and condition test phases.

The ECA rule execution model (rule execution models in general) has to address several issues. First, for various reasons multiple rules can be triggered and eligible for execution at the same time. For example, suppose that two rules r_i and r_j are defined, respectively, on two events E_1 and $(E_1 \nabla E_2)$ and there are no conditions specified for the rules, where $(E_1 \nabla E_2)$ is a disjunction of two component events E_1 and E_2. A disjunction occurs when either component event occurs [13]. Now, if event E_1 occurs, it will trigger both r_i and r_j. As addressed by Aiken et al. [3, 4], multiple triggered rules pose problems when different execution orders can produce different final database states. If an active database system randomly chooses a rule to execute (out of several triggered rules), as many extant systems do as the last resort, that will make the final database state *nondeterministic*. This adds to the problem of understanding applications that trigger rules.

To deal with multiple triggered rules, a generally taken approach is to define priorities among conflicting rules [2, 12, 28, 34]. When multiple conflicting rules are triggered at the same time, a rule with the highest priority is selected for execution. While we believe the prioritization is a sound approach, we notice that the previous priority schemes are incomplete and inadequate to handle the complexity caused by trigger relationships between rules.

On the other hand, Aiken et al. [3] focus on testing whether a given rule set has the *confluence* property. A rule set is said to be confluent if any permutation of the rules yields the same final database state when the rules are executed. If a rule set turns out to be not confluent, either the rule set is rewritten to remove the conflicts or priorities are explicitly defined over the conflicting rules. Then, the new rule set is retested to see if it has the confluence property. A problem with this approach is that it tends to repeat the time-consuming test process until the rule set eventually becomes confluent. Also, it has not shown by which mechanism confluence can be

guaranteed as priorities between conflicting rules are added to the system as a means of conflict resolution.

There are other subtle problems with the ECA rule execution model. Suppose that r_i and r_j mentioned previously have condition parts. As event E_1 occurs, the two rules pass the event detection phase. Assume that both rules have passed the condition test and are ready to execute their action part. It is possible that the execution of one rule's action, say r_i's, can invalidate r_j's condition that was already tested to be true; that is, r_i can *untrigger* r_j. Apart from the issue of whether or not the condition test should be delayed up to the point (or retested at the point) just before execution of the action part, if one rule untriggers other rules, it is very likely that the rule set is not confluent. The opposite situation can also happen. Suppose the condition of r_i was not met. So its action part would not be executed. But execution of r_j's action could change database state so that r_i's condition could be satisfied this time. Therefore, if r_j executes first and r_i's condition is tested after that, r_i will be able to execute too. Again, execution order of the two rules makes a difference. Instead of proposing a more rigorous rule execution model to deal with the anomalies, we consider such rules as conflicting with one another so that the rule designer can be informed of these rules. This view will allow us to cover the problem within the framework of confluent rule executions.

In this work we explore problems of confluent rule executions, which deal with obtaining a unique final database state for any initial set of rules that is triggered by a user transaction. We show that previous rule execution models are inadequate for confluent rule executions and propose a new rule execution model that guarantees confluent executions. It is also showed that our model is a perfect fit for parallel rule execution.

2.2 Limitations of the Earlier Rule Execution Models

Early rule execution models such as one used in OPS5 [9] deal with problems of confluent rule executions only in terms of conflict resolution. When multiple rules are triggered (and eligible for execution), the rule scheduler selects a rule to execute according to a certain set of criteria such as recency of trigger time, complexity of conditions. Although this scheme has been used in the AI domain, users in the database domain prefer to deterministic states from the executions of transactions. Furthermore, the conflict resolution approach is not a complete answer to the confluence problem since it is based on dynamic components such as recency of trigger time. For the above reasons, we do not consider this approach in our work.

A somewhat different approach taken in active database systems such as Starburst [2, 4] and Postgres [34, 35] is to statically assign execution priorities over rules. In these systems if multiple rules are triggered, a rule with the highest priority among them is executed first. However, rule execution models in these systems cannot guarantee confluent rule executions unless all the rules (not only conflicting ones) are totally ordered. This problem is illustrated in the following examples.

Example 2.2.1 Figure 2.1(a) shows a nondeterministic behavior of rule execution even when all conflicting rules are ordered. In the figure solid arrows represent trigger relationships. Dashed lines represent conflicts and an arrow on a dashed line indicates priority between two conflicting rules. As shown, two pairs of rules are conflicting: (r_2, r_5) and (r_3, r_5). The conflicting rules are ordered in such a way that r_2 precedes r_5 and r_5 precedes r_3 in execution when the pairs of conflicting rules are triggered at the same time. Now suppose r_1 and r_4 are triggered by the user transaction at the same time. (Note that these rules are denoted by solid circles in the graph.) In a rule execution model such as Starburst, one of r_1 and r_4 will be randomly picked for execution since there is no conflict between them, thus no order specified. Suppose r_4

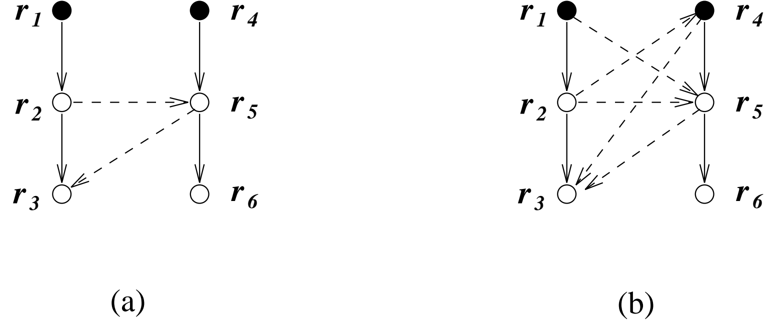

Figure 2.1. Rule execution graphs

is executed first; then it will trigger r_5. Yet there is no order between r_1 and r_5 which are ready to execute. So r_5 may go first, and its execution will trigger r_6. Then, r_6, r_1, r_2, and r_3 may follow. Including this execution sequence, two of all legitimate execution sequences for the rule set are as follows: (1) $\langle r_4 \cdot r_5 \cdot r_6 \cdot r_1 \cdot r_2 \cdot r_3 \rangle$ and (2) $\langle r_1 \cdot r_2 \cdot r_3 \cdot r_4 \cdot r_5 \cdot r_6 \rangle$. Note that relative orders of two conflicting rules, r_2 and r_5 (as well as r_3 and r_5) in the two rule execution sequences are different, thereby unable to guarantee confluent execution. $\lhd$

Example 2.2.2 Figure 2.2 illustrates another situation where the previous rule execution models fail to achieve confluent rule executions. There is a dependency between r_k and r_l, and r_k has priority over r_l. In this example, r_i and r_j are triggered by the user transaction. Note also that r_i is an ancestor of r_j in the trigger relationship and thereby trigger paths originated from both rules overlap one another. Given that priority, the following two (and more) sequences of rule executions are possible: $\langle r_i \cdot r_j \cdot r_k \cdot r_l \cdot r_j \cdot r_k \cdot r_l \rangle$ and $\langle r_i \cdot r_j \cdot r_k \cdot r_j \cdot r_k \cdot r_l \cdot r_l \rangle$. Now relative orders of two conflicting rules r_k and r_l in the two execution sequences are different. Therefore confluent rule executions cannot be guaranteed in the given situation. $\lhd$

As the previous examples suggest, a problem that the extant active rule execution models fail to address properly is that even though two rules are not directly conflicting each other, they may trigger other rules that are directly conflicting. Depending

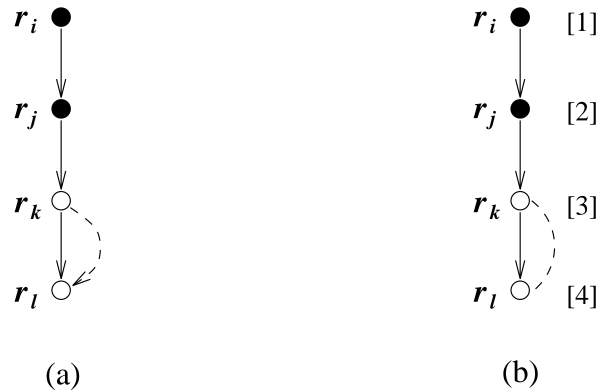

Figure 2.2. Overlapped trigger paths

on execution order of triggering rules, the directly conflicting rules may be executed in a different order from what the user specified, likely resulting in non-confluent rule executions. Unless all the direct conflicts are removed by rewriting the rules, one possible remedy for this problem, implied in Starburst [3], would be to regard the indirectly conflicting rules as conflicting ones. Figure 2.1(b) illustrates how conflicts of Figure 2.1(a) are propagated toward ancestor rules in the trigger relationship as this approach is taken. An undesirable consequence of propagating conflicts is that it severely limits parallel rule execution. In addition, it is not always clear how to propagate conflicts in some cases as Figure 2.2(a) shows.

Another problem that the previous rule execution models do not handle was shown in Example 2.2.2 where trigger paths of rules triggered by the user transaction overlap. In fact, this new situation poses additional problems for priority specification. That is, any static priority schemes specified before rules' execution cannot range over all possible permutations of conflicting rules execution, since one cannot anticipate which rules will be triggered by the user transaction how many times. For instance, given the rule set of Figure 2.2(a), there can be two distinct final database states which result from rule execution sequences, $\langle r_i \cdot r_j \cdot r_k \cdot r_l \cdot r_j \cdot r_k \cdot r_l \rangle$ and $\langle r_i \cdot r_j \cdot r_k \cdot r_j \cdot r_k \cdot r_l \cdot r_l \rangle$. All other legitimate rule execution sequences are equivalent to one of the two sequences

in terms of final database states. However, if r_j is triggered twice and r_i once by the user transaction, the number of distinct final database states increases up to five. As r_i and r_j are triggered more number of times, the number of different final database states increases exponentially. Therefore, it is not realistic to provide every possible alternatives for these cases. Rather, a less general scheme of priority specification, which provides only some specific alternatives, needs to be considered.

Figure 2.2(b) shows one way of specifying priority for the rule set of Figure 2.2(a), which is similar to the priority scheme adopted in Postgres [34]. Numbers in brackets denote absolute priorities associated with rules. A larger number denotes a higher priority. This priority specification guarantees confluent rule executions although non-conflicting rules (r_i and r_j) too need to be assigned priorities. Note that the given priority specification is (unnecessarily) so strong that it effectively imposes a serial execution order $\langle r_j \cdot r_k \cdot r_l \cdot r_i \cdot r_j \cdot r_k \cdot r_l \rangle$, thereby ruling out any parallel rule executions. For instance, one instance of r_j could run in parallel with a series of r_i and r_j without affecting the final database state.

In the subsequent sections, we develop a novel rule execution model and a priority scheme that not only ensures confluent rule executions but also allows greater parallelism.

2.3 Assumptions and Definitions

2.3.1 Rule Execution Sequence (RES) and Rule Commutativity

Informally, a rule execution sequence (RES) is a sequence of rules that the system can execute when a user transaction triggers at least one rule in the sequence. To characterize RESs, we first define partial RESs. Throughout this work, R denotes *system rule set*, a set of rules defined in the system by the user. D denotes a set of all possible database states determined by the database schema. (d_j, R_k), $d_j \in D$ and $R_k \subseteq R$, denotes a pair of a database state and a triggered rule set. If R_k is a

set of rules directly triggered by a user transaction, it is specially called *UTRS* that stands for User-Triggered-Rules-Set. UTRS is, in fact, a multiset since as we shall see later, multiple instances of a rule can be in it. $\mathcal{S}$ denotes a set of all partial RESs (see below) defined over R and D.

Partial RES. Given R and D, for a nonempty set of triggered rules, $R_k \subseteq R$ and a database state $d_j \in D$, a *partial RES*, σ is defined to be a sequence of rules that connects pairs of a database state and a triggered rule set as follows:

$$\sigma = \langle (d_j, R_k) \xrightarrow{r_i} (d_{j+1}, R_{k+1}) \xrightarrow{r_{i+1}} \cdots \xrightarrow{r_{i+m-1}} (d_{j+m}, R_{k+m}) \rangle$$

where $d_{j+l} \in D$ $(1 \leq l \leq m)$ is a new database state obtained by execution of r_{i+l-1}, each rule r_{i+l} $(0 \leq l < m)$ is in a triggered rule set R_{k+l}, and eligible for execution in d_{j+l}, i.e., d_{j+l} evaluates the rule's condition test to true. Each triggered rule set $R_{k+l} \subseteq R$ $(1 \leq l \leq m)$ is built as $R_{k+l} = ((R_{k+l-1} - \{r_{i+l-1}\}) - Ru_{k+l}) \cup Rt_{k+l}$, where Ru_{k+l} is a set of rules untriggered by r_{i+l-1} and Rt_{k+l} is a set of rules triggered by r_{i+l-1}.[2] $\lhd$

In this work, if only sequences of rule executions are of interest, for simplicity we write a partial RES without associated database states and triggered rule sets. For example, the partial RES above can be denoted as $\sigma = \langle r_i \cdot r_{i+1} \cdots r_{i+m-1} \rangle$, and we already used this form of partial RESs in the previous sections.

Among partial RESs, we are interested in some, called complete RESs (or simply, RESs), that satisfy certain conditions.

Complete RES. Given R and D, for a nonempty set $R_k \subseteq R$ that is a set of rules triggered by a user transaction (i.e., UTRS) and $d_j \in D$ is a database state produced

[2] Subscripts, $i, i+1, \cdots, i+m-1$, attached to rules, are intended to mean that they are m rules that need not be distinct (similarly for d's and R's). They do not represent any sequential order of rules with respect to subscript numbers. That is, they should not be interpreted as, for instances, $r_{10}, r_{11}, r_{12} \cdots$, in case where r_i is r_{10}. For a precise denotation, we could use $i_0, i_1, \cdots, i_{m-1}$, instead. However, we have opted for the less precise notation in favor of simplicity throughout this work.

by operations in the user transaction, a *complete RES* (or *RES*), σ is defined to be a partial RES:

$$\sigma = \langle (d_j, R_k) \xrightarrow{r_i} (d_{j+1}, R_{k+1}) \xrightarrow{r_{i+1}} \cdots \xrightarrow{r_{i+m-1}} (d_{j+m}, R_{k+m} = \emptyset) \rangle$$

where no triggered (and eligible) rules remain after execution of the last rule r_{i+m-1} (i.e., $R_{k+m} = \emptyset$). $\triangleleft$

Note that given R_k and d_j, there may be multiple different RESs, even in a case where there is only one rule in R_k, and those RESs do not necessarily have the same set of rules executed, since a rule's triggering/untriggering other rules may be dependent on the current database state. In this work we use *rule schedule* in informal settings interchangeably with complete RES.

Rule shuffling. Given a partial RES σ_1, two rules r_i and r_j in σ_1 can exchange their positions provided $r_j \in R_y$, yielding a different partial RES σ_2 as below:

$$\sigma_1 = \langle (d_x, R_y) \xrightarrow{r_i} (d_k, R_l) \xrightarrow{r_j} (d_u, R_v) \rangle$$

$$\sigma_2 = \langle (d_x, R_y) \xrightarrow{r_j} (d_m, R_n) \xrightarrow{r_i} (d_s, R_t) \rangle$$

$\triangleleft$

Next we define an important property of rules that is used to show if a system rule set is confluent. Two rules are defined to be commutative if shuffling them always yields the same result.

Rule commutativity. Given R and D, two rules $r_i, r_j \in R$ are defined to be *commutative*, if for all $R_y \subseteq R$, where $r_i, r_j \in R_y$, and for all database state $d_x \in D$, the following two partial RESs can be defined:

$$\langle (d_x, R_y) \xrightarrow{r_i} (d_k, R_l) \xrightarrow{r_j} (d_u, R_v) \rangle$$

$$\langle (d_x, R_y) \xrightarrow{r_j} (d_m, R_n) \xrightarrow{r_i} (d_u, R_v) \rangle$$

where $d_x, d_k, d_m, d_u \in D$ need not be distinct and likewise $R_y, R_l, R_n, R_v \subseteq R$ need not be distinct. $\lhd$

Note that any rule is trivially commutative to itself.

Equivalent partial RESs. Two partial RESs σ_i and σ_j are defined to be *equivalent* ($\equiv$) if:

1. σ_i and σ_j begin with the same pair of database state and triggered rule set, and end with the same pair of database state and triggered rule set; and
2. in σ_i and σ_j the same set of rules is triggered, possibly in different orders. $\lhd$

The two partial RESs shown in the definition of rule commutativity are equivalent. In fact, the rule commutativity is used to prove that two or more partial RESs are equivalent, and the equivalence of partial RESs is, in turn, used to show whether given a system rule set is confluent or not. Incidentally, it should be noted that without condition 2), the equivalence definition can still be valid. However, with our static analysis method it is not possible to identify all such equivalent partial RESs. To make presentation of this work coherent, we chose a more restrictive form of equivalence.

The equivalence of partial RESs naturally lends itself to definition of equivalence classes of partial RESs. For given R and D, the set of all partial RESs, $\mathcal{S}$ is partitioned into disjoint classes by the equivalence relation ($\equiv$). All partial RESs belonging to an equivalence class have the same final result, i.e., the same database state and the triggered rule set.

Equivalence class of partial RESs. For a partial RES, $\sigma \in \mathcal{S}$, the *equivalence class* of σ is the set S_σ defined as follows:

$$S_\sigma = \{\gamma \in \mathcal{S} \mid \gamma \equiv \sigma\}.$$

$\lhd$

Of partial RESs belonging to the same equivalence class, for the discussion in this work we define *canonical partial RES*, or *canonical RES* for short, to be a partial RES that comes first when all the partial RESs are sorted by their rules' concatenated subscripts in lexicographical order. For instance, assuming that an equivalence class includes only three partial RESs, $\sigma_i = \langle r_1 \cdot r_2 \cdot r_4 \cdot r_3 \rangle$, $\sigma_j = \langle r_1 \cdot r_4 \cdot r_2 \cdot r_3 \rangle$, and $\sigma_k = \langle r_1 \cdot r_2 \cdot r_3 \cdot r_4 \rangle$, σ_k is the canonical RES representing that equivalence class (by lexicographically sorting concatenated subscripts of the partial RESs, i.e., 1234 (σ_k), 1243 (σ_i), and 1423 (σ_j)) Of our prime interest is the equivalence class of complete RESs.

Confluent rule set. Given R and D, if there exists only one equivalence class of complete RESs for every nonempty set $R' \subseteq R$ and every $d \in D$, R is defined to be *confluent*. $\lhd$

2.3.2 Dependencies and Dependency Graph

If a different execution sequence of the same rules can produce a different final database state, it is because of certain interactions between rules and between rules and the environment. If we assume the execution environment to be fixed and there is no interference from the user while rules are executing, the interactions between rules must be the sole reason for non-confluent rule executions. Based on this, we define rules' interactions responsible for non-confluence as *dependencies* between rules, much like those of the concurrency control in transaction processing. We define two kinds of dependencies.

Data dependency. Two distinct rules r_i and r_j are defined to have *data dependency* with each other if r_i writes in its action part to a data object that r_j reads or writes in its action part, or vice versa. ◁

Untrigger dependency. Two distinct rules r_i and r_j are defined to have *untrigger dependency* with each other if r_i writes in its action part to a data object that r_j reads in its condition part, or vice versa. ◁

If two rules have data dependency with each other, the input to one rule can be altered by the other rule's action. Thus it is very likely that the affected rule would behave differently. The data dependency can also mean that one rule's output can be overridden by the other rule's output. This also has a bearing on the final outcome. If there is no data dependency, two rules act independently. Therefore, there should be no difference in the final outcome due to a different relative execution order of the two rules.

On the other hand, if there is untrigger dependency between two rules r_i and r_j, this implies that one rule's action can change the condition which determines whether the other rule is to execute or not. If the affected rule, say r_i, has already executed first, it is unrealistic to revoke the effect of r_i. As a result, both r_i and r_j will execute in this case. However, if the affecting rule r_j executes first, it can prevent r_i from executing. Since it is assumed that there are no read-only rules, the two different execution sequences can result in different database states even though there is no data dependency.[3]

From the observation above, it is clear that the absence of data dependency and untrigger dependency between two rules is a sufficient condition for the two rules to be

[3]It should be noted that whether or not the untrigger dependency can indeed affect confluent execution depends on rule execution model employed by an active database system. If the rule execution model does not re-check the condition part of a rule just before it executes the action part of that rule, then no rule is untriggered. In such a case, it can appear that the untrigger dependency is no longer a problem and only data dependency matters.

commutative. (The reverse is not necessarily true.) If there exists either dependency between two rules, the rules are said to *conflict* with each other. Obviously, conflicting rules are non-commutative.

Lemma 1 *Given a partial RES σ, a new partial RES σ' obtained by freely shuffling rules in σ is equivalent to σ, as long as relative orders of conflicting rules in both RESs are equal if there are any conflicting rules.*

Proof of Lemma 1 Suppose σ and σ' are not equivalent despite the same relative orders of conflicting rules in them. Then, there must be one or more pairs of non-conflicting rules in σ that can be shuffled but result in a different (non-equivalent) partial RES. These non-conflicting rules are, then, conflicting and should have the same relative orders in σ and σ', which is a contradiction. $\square$

Below, we define dependency graph that represents dependencies between rules in the system rule set.

Dependency graph. Given system rule set R, a *dependency graph*, $DG = (R, E_D)$ is an undirected graph where R is a node set in which a node has one-to-one mapping to a rule and E_D is a dependency edge set. For any nodes u and v in R, a *dependency edge* (u, v) is in E_D if and only if there is data dependency or untrigger dependency (or both) between u and v. $\triangleleft$

A dependency graph is non-transitive; that is, (u, v) and (v, w) in E_D do not imply (u, w) in E_D. Edges in a dependency graph represent only direct dependencies. An indirect (transitive) dependency is represented by a path consisting of a set of connected dependency edges.

2.3.3 Trigger Graph

A *trigger graph* (TG) is an *acyclic* directed graph representing trigger relationships between rules within a given system rule set R. For system rule set R, $TG = (R, E_T)$

has a node set R in which a node has one-to-one mapping to a rule and a trigger edge set E_T. For any two nodes (i.e., rules) r_i and r_j in R, trigger edge set E_T contains a directed edge, called *trigger edge*, $(r_i \xrightarrow{T} r_j)$, if and only if r_i *can trigger* r_j. It is defined that r_i can trigger r_j if execution of r_i's action can give rise to *an* event that is referenced in the event specification of r_j.[4] A *trigger path* in a trigger graph is a (linear) path starting from any node and ending at a reachable leaf node.

Note that for rules r_i and r_j above, it is possible that r_j is not triggered by r_i at run time if r_i's action part contains a conditional statement. Nevertheless we conservatively maintain a trigger edge if there is any possibility of r_i's triggering r_j. In addition, we are assuming that a trigger graph is *acyclic* to guarantee termination of rule executions [3]. If a trigger graph contains a cycle, it is possible that once a rule in the cycle is triggered all the rules in the cycle keep triggering the next rule indefinitely. We also assume that there exists a separate mechanism for detecting cycles in a trigger graph so that the rule designer can rewrite the rules in such a case.

Incidentally, it should be noted that a trigger relationship between two rules does not necessarily imply a dependency between the rules. For instance, given a trigger edge $(r_i \xrightarrow{T} r_j)$, if r_i for sure triggers r_j and no other rules are triggered from r_i and r_j, there are only two possible partial RESs for the two rules, $\langle r_i \cdot r_j \cdot r_j \rangle$ and $\langle r_j \cdot r_i \cdot r_j \rangle$. If there is no data or untrigger dependency between r_i and r_j (i.e, the two rules are commutative), the two RESs are equivalent despite the trigger edge.

2.4 Confluence and Priority Specification

In this section we present basic ideas that give us a handle for dealing with conflicting rules in order to obtain confluent rules executions. We consider simple

[4]We admit that this definition of "can trigger" is rather crude. In Sentinel, for example, if a rule is waiting for an occurrence of $(E_1; E_2)$, which is a composite event *sequence* and occurs when E_2 occurs provided E_1 occurred before, the occurrence of E_1 alone never triggers that rule. In our current work, however, we do not pursue this issue any further. (For event specifications in Sentinel, see Chakravarthy et al. [13].)

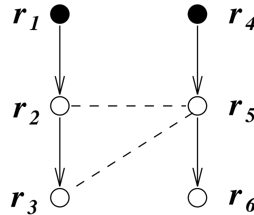

Figure 2.3. A conflicting rule set

cases first. When there are n distinct rules to execute and m pairs of conflicting rules among them, intuitively, the maximum number of different final database states that can result from all differing RESs is conservatively bounded by 2^m, since each pair of conflicting rules can possibly produce two different final database states by changing their relative order.

Example 2.4.1 Figure 2.3 is a redrawing of Figure 2.1(a) with removal of directions on dependency edges (r_2, r_5) and (r_3, r_5). Note that r_1 and r_4, denoted by solid nodes in the graph, are in UTRS, a set of rules initially triggered by a user transaction. Assuming that all the six rules are executed, all complete RESs that can be generated should be equal to a set of possible merged sequences of two partial RESs $\langle r_1 \cdot r_2 \cdot r_3 \rangle$ and $\langle r_4 \cdot r_5 \cdot r_6 \rangle$. Then, all the possible merged (now complete) RESs can be partitioned into up to four groups by relative orders between r_2 and r_5 and between r_3 and r_5 as follows: (1) $(r_2 \rightarrow r_5)$ $(r_3 \leftarrow r_5)$, (2) $(r_2 \rightarrow r_5)$ $(r_3 \rightarrow r_5)$, (3) $(r_2 \leftarrow r_5)$ $(r_3 \leftarrow r_5)$, and (4) $(r_2 \leftarrow r_5)$ $(r_3 \rightarrow r_5)$. However, since there exists an inherent order between r_2 and r_3, i.e., $(r_2 \rightarrow r_3)$, dictated by a trigger relationship, no merged RESs can contain combination (4) due to a cycle being formed. Combination (4) is dropped from consideration. Since cumulative effect of all the other rules are the same regardless of their execution order, the three combinations are the only factor that can make a difference in the final database state. Therefore, in this example, up to three distinct final database states can be produced by all possible complete RESs. $\lhd$

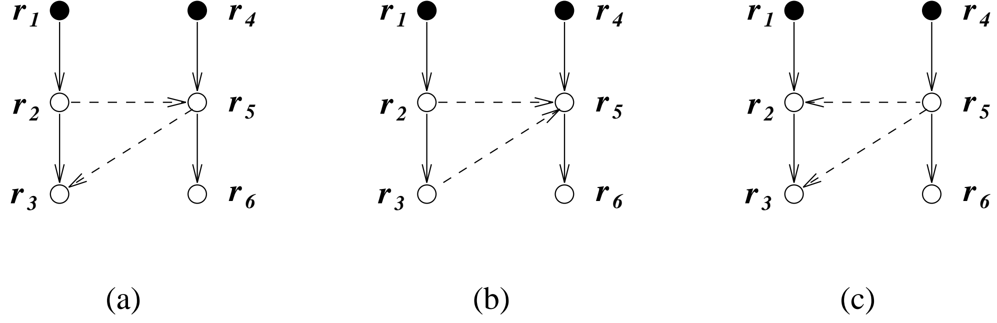

Figure 2.4. Priority graphs for Figure 1.4

Using the three possible orderings of conflicting rules in Example 2.4.1, we can assign directions to dependency edges in the graph of Figure 2.3. Resulting graphs, which we call *priority graphs*, are shown in Figure 2.4. These priority graphs present how priorities can be specified over conflicting rules in order to make rule executions confluent. Also, importantly, they represent partial orders that the rule scheduler needs to follow as it schedules rule executions. As we shall see in the following section, the rule scheduler basically uses a topological sort algorithm working on a subgraph of priority graph, and this demands the priority graph to be *acyclic*.

Example 2.4.2 All possible topological sorts on priority graph of Figure 2.4(a) correspond to an equivalence class represented by a canonical RES, $\sigma_1 = \langle r_1 \cdot \bar{r_2} \cdot r_4 \cdot \bar{r_5} \cdot \bar{r_3} \cdot r_6 \rangle$ – for clarity we use $^-$ hereafter to denote conflicting rules as $\bar{r_2}$. Note that a RES, $\langle r_1 \cdot r_4 \cdot \bar{r_2} \cdot \bar{r_5} \cdot \bar{r_3} \cdot r_6 \rangle$ is equivalently converted to σ_1 by shuffling r_2 and r_4, which are commutative. Similarly, $\sigma_2 = \langle r_1 \cdot \bar{r_2} \cdot \bar{r_3} \cdot r_4 \cdot \bar{r_5} \cdot r_6 \rangle$ and $\sigma_3 = \langle r_1 \cdot r_4 \cdot \bar{r_5} \cdot \bar{r_2} \cdot \bar{r_3} \cdot r_6 \rangle$ represent equivalence classes obtained when the topological sort is carried out on priority graphs of Figures 2.4(b) and (c), respectively. $\lhd$

The formal definition of the priority graph is given below.

Priority graph. Given trigger graph $TG = (R, E_T)$ and dependency graph $DG = (R, E_D)$, *priority graph*, $PG = (R, E_P)$ is a directed *acyclic* multigraph formed

by merging TG and DG, where R is a node set defined as before and E_P is a priority edge set. For any two distinct nodes $u, v \in R$, $(u \xrightarrow{T} v) \in E_P$ if and only if $(u \xrightarrow{T} v) \in E_T$, and either $(u \xrightarrow{D} v) \in E_P$ or $(v \xrightarrow{D} u) \in E_P$ if and only if $(u, v) \in E_D$. $(u \xrightarrow{D} v)$ (or $(v \xrightarrow{D} u)$) is called *directed dependency edge* to distinguish it from undirected ones in E_D, and the direction of the edge is given by the user. $\lhd$

For a PG, a trigger edge is depicted by a solid arrow line while a directed dependency edge is depicted by a dashed arrow line. A PG is defined to be a multigraph because it can have more than one edge (actually two edges, i.e., a trigger edge and a directed dependency edge) between two nodes. It should be noted that if a node u is an ancestor of a node v in TG and there is a dependency edge (u, v) in DG, the corresponding directed dependency edge in PG is automatically set to $(u \xrightarrow{D} v)$, not to form a cycle in PG.

Example 2.4.3 Figures 2.5(a) and (b) show a trigger graph and its dependency graph counterpart respectively. Figure 2.6 shows a priority graph built out of the two graphs. Note that directions of dependency edges are determined by the user as a way of specifying priorities between conflicting rules. However, direction of dependency edge between r_1 and r_8 (between r_2 and r_4 as well) is set by the system as shown in the graph because r_1 is an ancestor of r_8 in a trigger path. $\lhd$

Note that given a system rule set R, when an active database system is running, subsets of R will be triggered and executed dynamically. In order to schedule those rules, the rule scheduler builds a subgraph of a given PG, called *execution graph*, when a user transaction triggers rules.

Execution graph. Given a system rule set R, a priority graph $PG = (R, E_P)$, and a UTRS $R' \subseteq R$, an *execution graph* $EG = (R_E, E_E)$ is a subgraph of PG where R_E is a node set and E_E is an edge set. R_E is recursively defined as $R_E = \{r \mid$

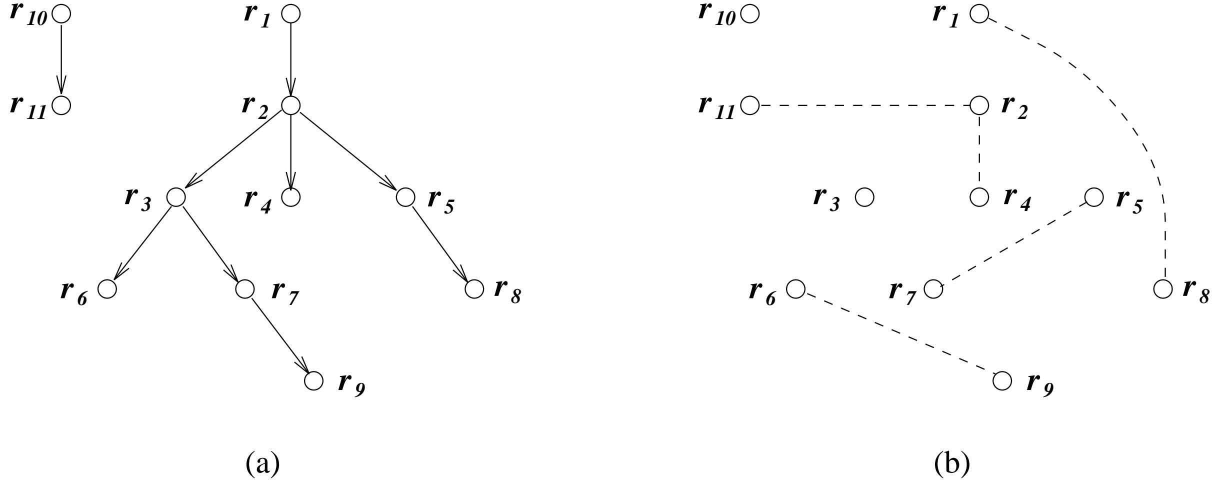

Figure 2.5. A pair of trigger graph and dependency graph

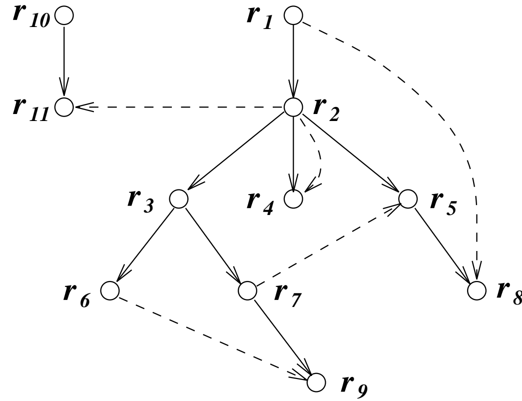

Figure 2.6. A priority graph

$r \in R' \vee (\exists r' \exists (r' \xrightarrow{T} r)\ (r' \in R_E \wedge (r' \xrightarrow{T} r) \in E_P)\}$. For any two distinct nodes $u, v \in R_E$, $(u \xrightarrow{T} v) \in E_E$ if $(u \xrightarrow{T} v) \in E_P$ and $(u \xrightarrow{D} v) \in E_E$ if $(u \xrightarrow{D} v) \in E_P$. $\lhd$

Simply stated, the node set R_E consists of a UTRS plus those rules that are reachable from rules in the UTRS through trigger paths in PG. The edge set E_E is a set of trigger and directed dependency edges that connect nodes in R_E.

Example 2.4.4 Figure 2.7 shows an execution graph derived from a priority graph of Figure 2.6 when a UTRS has rules r_3, r_5, and r_{10}. A rule schedule can be obtained by performing the topological sort on the execution graph. The canonical RES for the

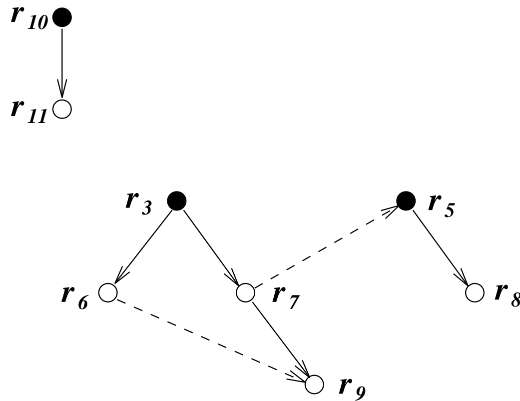

Figure 2.7. An execution graph

equivalence class represented by the execution graph is $\langle r_3 \cdot r_6 \cdot r_7 \cdot r_5 \cdot r_8 \cdot r_9 \cdot r_{10} \cdot r_{11} \rangle$. Note that priority graphs shown in Figure 2.4 are, in fact, execution graphs as well. $\triangleleft$

Lemma 2 *Given execution graph $EG = (R_E, E_E)$, a set of rule execution sequences corresponding to all feasible topological sorts on EG constitutes an equivalence class, independent of initial database state.*

Proof of Lemma 2 Since EG is acyclic and all pairs of conflicting rules in EG are ordered (i.e., have an edge between them), all topological sorts on EG should have the same relative orders of conflicting rules. Then, by Lemma 1, RESs represented by the topological sorts are equivalent to each other. Also, since Lemma 1 holds without any premise on initial database states, Lemma 2 also holds regardless of initial database states. $\square$

The power of choice is now given to the users. There may be one system-wise priority graph for all rules defined in the system. All applications will be governed by a single type of confluent rule executions in such a case. More preferably, however, each user (or each application) may have a separate priority graph tailored for specific

needs. Also, given a conflicting rule set, the user may choose to specify priorities over only a part of conflicting rules and is not concerned about the rest of them. The rule scheduler will ignore unspecified (thus undirected) dependency edges in a priority. Taking this approach, our rule execution model can readily meet various user requirements with respect to confluent rule executions.

CHAPTER 3
IMPLEMENTATION OF CONFLUENT RULE SCHEDULER

3.1 Strict Order-Preserving Rule Execution Model

In the previous chapter, only simple cases were considered. In particular, the structures of trigger graphs were trees, all rules in a UTRS were distinct and no trigger paths existed between these rules. As a result, no trigger paths in the execution graph overlapped with one another. When rules in a UTRS have overlapping trigger paths, the priority graph and execution graph defined in the previous chapter do not capture the semantics. For example, consider Figure 3.1(a) which is the same as Figure 2.2(a). When r_i and r_j are triggered by a transaction, both rules instantiate their own trigger paths, and these trigger paths overlap with each other.[1] If we think of the graph as an execution graph, it yields two partial RESs from the graph: $\langle r_i \cdot r_j \cdot \bar{r_k} \cdot \bar{r_l} \rangle$ and $\langle r_j \cdot \bar{r_k} \cdot \bar{r_l} \rangle$. Therefore, a rule schedule (alternatively a complete RES) should be a merged RES of the two partial RESs. A possible merged RES is $\langle r_i \cdot r_j \cdot r_j \cdot \bar{r_k} \cdot \bar{r_l} \cdot \bar{r_k} \cdot \bar{r_l} \rangle$. Issues to be addressed in this chapter are, (i) obtaining rule schedules from an execution graph where trigger paths are overlapping, (ii) assurance of the confluence property when rules are executed in accordance with any of such rule schedules, and (iii) parallel rule schedule taking advantage of the availability of confluent multiple rule schedules.

3.1.1 Extended Execution Graph

In order to understand the effect of overlapping trigger paths, we introduce an extended execution graph, used only for illustration purposes. Recall that given a

[1] The overlapping of trigger paths is not directly visible in Figure 3.1(a).

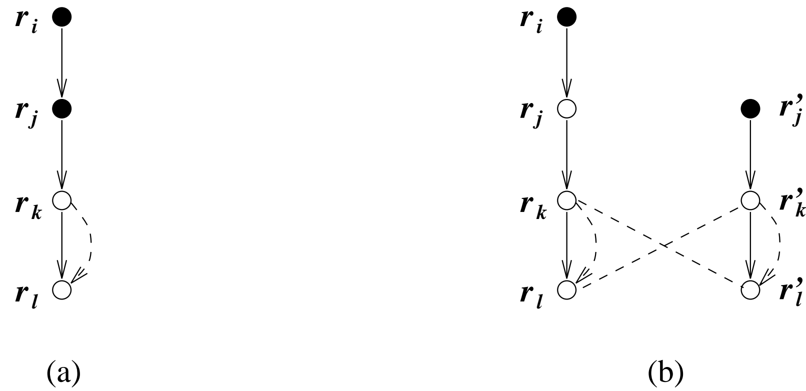

Figure 3.1. Overlapping trigger paths and extended execution graph

system rule set along with its trigger graph, each rule in a UTRS instantiates a subgraph of the trigger graph, whose sole root is that rule. However, it is possible to derive an execution graph, from given a priority graph and a UTRS, such that every rule in the UTRS becomes *a* root node in the resulting extended execution graph.

Example 3.1.1 Figure 3.1(b) shows the extended execution graph of Figure 3.1(a). In the extended execution graph, r_j and r'_j (similarly other rules as well) are the same rule and only represent different instantiations. Since there is a dependency between rules r_k and r_l, this dependency may well be present between all instantiations of r_k and r_l as shown in Figure 3.1(b). Directions of dependency edges in an extended execution graph might be either inferred from the priority graph or specified by the user. Figure 3.2 shows three different acyclic orderings of relevant dependency edges of Figure 3.1(b). Once an acyclic extended execution graph is given, the rule scheduler can schedule rule executions using the topological sort. All possible topological sorts constitute one equivalence class. $\lhd$

The extended execution graph, however, *cannot* be used for priority specification; it is created only at rule execution time and priorities need to be specified before that. Thus we need to find alternative ways to interpret a priority graph.

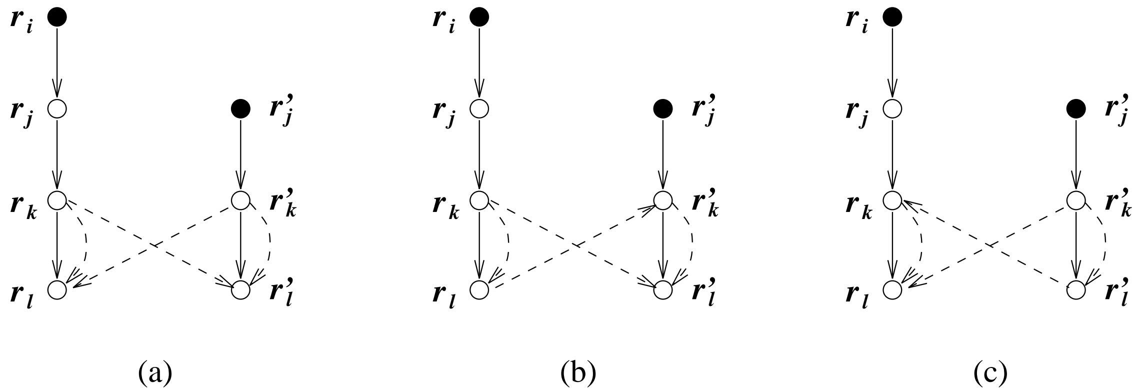

Figure 3.2. Three different orderings of dependency edges in Figure 3.1(b)

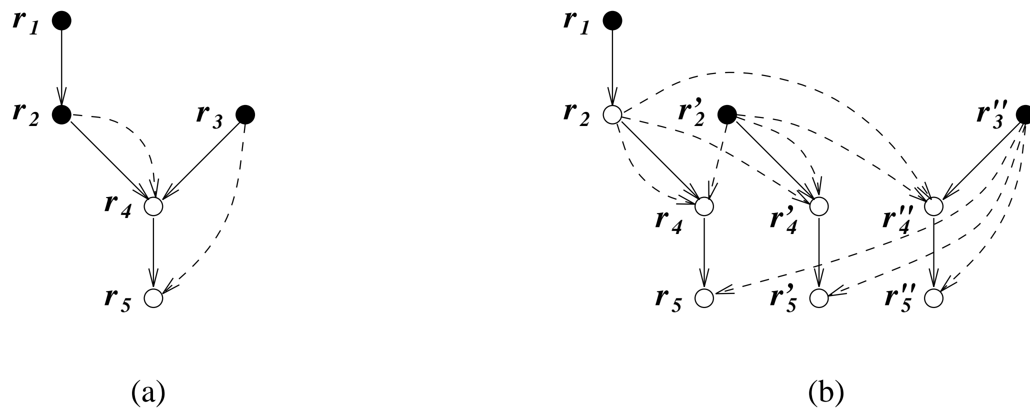

Figure 3.3. Extended execution graph in strict order-preserving executions

3.1.2 Strict Order-Preserving Executions

One way to derive an extended execution graph is to faithfully follow what the user specifies in the priority graph, i.e., priorities between conflicting rules. In *strict order-preserving executions*, if rule r_i has precedence over rule r_j in a priority graph, all instances of r_i precede all instances of r_j in resulting rules schedules. Given a priority graph, an extended execution graph is obtained by simply adding directed dependency edges in the priority graph to duplicated overlapping trigger paths. This scheme provides a simple solution for overlapping trigger paths, regardless of the number of times trigger paths overlap.

Example 3.1.2 Figure 3.3(a) shows a priority graph where rules r_1, r_2, and r_3 are in UTRS and their overlapping trigger paths are denoted by partial RESs, $\langle r_1 \cdot \bar{r_2} \cdot \bar{r_4} \cdot \bar{r_5} \rangle$, $\langle \bar{r_2} \cdot \bar{r_4} \cdot \bar{r_5} \rangle$, and $\langle \bar{r_3} \cdot \bar{r_4} \cdot \bar{r_5} \rangle$. Figure 3.3(b) illustrates how an extended execution graph is built using strict order-preserving executions. First, overlapping trigger paths are separated. Second, any dependency edge in the priority graph that connects a rule in an overlapping trigger path and a rule in any trigger path is also introduced in the extended execution graph. For example, $(r_2 \xrightarrow{D} r_4')$ and $(r_3'' \xrightarrow{D} r_5')$. Given the extended execution graph, the rule scheduler can schedule rule execution by performing the topological sort. All feasible topological sorts should constitute one equivalence class since in the topological sorts, executions of all conflicting rules are ordered in the same ways. The canonical RES for the extended execution graph of Figure 3.3(b) is $\langle r_1 \cdot \bar{r_2} \cdot \bar{r_2} \cdot \bar{r_3} \cdot \bar{r_4} \cdot \bar{r_4} \cdot \bar{r_4} \cdot \bar{r_5} \cdot \bar{r_5} \cdot \bar{r_5} \rangle$. (Note that r_i, r_i', and r_i'' are the same rule.) $\triangleleft$

3.1.3 Implementation

In order to implement a rule scheduler conforming to the strict order-preserving execution, one could directly use extended execution graphs such as Figure 3.3(b). However, there is a simpler way to derive an execution graph without having to duplicate every overlapping trigger path. Consider the extended execution graph of Figure 3.3(b). In the strict order-preserving execution, directions of all dependency edges incoming to and outgoing from overlapping trigger paths are all the same as shown in the graph. Therefore, it is unnecessary to duplicate overlapping trigger paths. We, instead, add a rule_count to each node of a plain execution graph. A rule_count attached to a node indicates how many rules in UTRS share the trigger path which the node (i.e., rule) belongs to. Figure 3.4 depicts how the plain execution graph of Figure 3.3(a) is extended using rule_counts. In the new graph, $m(i)$ represents that the trigger path is shared by i instances of the associated rule.

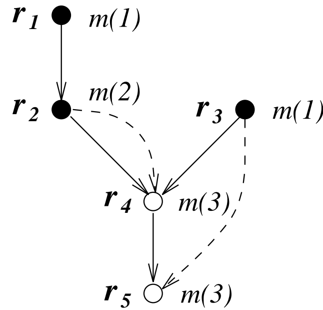

Figure 3.4. Extended execution graph with rule_counts

Now the new extended execution graph can be used with a minor modification to the topological sort. Whenever a rule is scheduled, its rule_count in the execution graph is decreased by 1. If the rule_count reaches 0, the node and outgoing edges are removed from the execution graph.

Figure 3.5 describes an algorithm, *Build_EG()* for building an execution graph for given a priority graph (PG) and a UTRS. It uses $M[\,]$, an array of size of the system rule set R, to hold rule_count values of rules in EG. If rule r_i is (to be) in EG, $M[i]$ represents the rule_count attached to r_i. *Build_EG()* calls a subroutine, *DFS_Copy_Tree()* for every instance in UTRS. Remember that UTRS is a multiset. It can have multiple instances of the same rule due to multiple triggers. *DFS_Copy_Tree()* traverses the PG in the depth-first search fashion and copies a portion of the PG that are reachable through trigger edges in PG. It also increases rule_count of each node that it visits. The execution graph of Figure 3.4 is obtained by applying *Build_EG()* to the priority graph of Figure 3.3(a) with $UTRS = \{r_1, r_2, r_3\}$.

Once an execution graph is built by *Build_EG()*, the rule scheduler can schedule rule executions. However, it is possible that some rules in a trigger path in an execution graph are not triggered at all by its parent rule at run time. In such a case, other rules in other trigger paths that have an incoming dependency edge from that rule should not wait. Otherwise the rule scheduler will get stuck. Furthermore,

```
Given $PG$ and $UTRS$:

Build_EG()
{
$EG = \emptyset$;
initialize array M[ ] to 0's; // M[ ] is rule_count array
for every $r_i \in UTRS$ do
    call DFS_Copy_Tree($r_i$);
}


DFS_Copy_Tree($r_i$)
{
if $(r_i \notin EG)$ then
    copy $r_i$ into $EG$;
M[i]++; // increase rule_count of node $r_i$
// copy trigger edges
for all $r_j$ such that $\exists(r_i \xrightarrow{T} r_j) \in PG$ do
    {
    call DFS_Copy_Tree($r_j$);
    if $((r_i \xrightarrow{T} r_j) \notin EG)$ then
        copy $(r_i \xrightarrow{T} r_j)$ into $EG$;
    }
// copy dependency edges
for all $r_j$ such that $\exists(r_i \xrightarrow{D} r_j) \in PG$ do
    {
    if $(r_j \in EG)$ and $((r_i \xrightarrow{D} r_j) \notin EG)$ then
        copy $(r_i \xrightarrow{D} r_j)$ into $EG$;
    }
}
```

Figure 3.5. Algorithm – Build_EG()

if a rule is not triggered, all its descendant rules in trigger path will not be triggered either. This consideration should be taken recursively downward trigger paths.

Figure 3.6 describes the rule scheduling algorithm, *Schedule()*, which is a modified version of topological sort. Given an execution graph EG, it arbitrarily selects a node (i.e., rule) with indegree 0. Since EG is acyclic, there should always be at least one node with indegree 0. After executing the selected rule, the scheduler decreases rule_count of the node by 1, and if the rule_count reaches 0, the node is removed along with any trigger and dependency edges outgoing from the node. However, before the removal, it checks whether the executed rule has triggered child rules in trigger paths. If there are child rules that are not triggered, then *Schedule()* calls a subroutine *DFS_Dec_M()* for those child rules. *DFS_Dec_M()* traverses down EG in a depth-first search fashion and decreases rule_count of each node it visits by 1. If rule_count of a node becomes 0, it removes the node and all outgoing edges.

Theorem 1 *The strict order-preserving rule execution model guarantees confluent rule executions.*

Proof of Theorem 1 Based on Lemma 2, algorithms *Build_EG()* and *Schedule()* together serve as a constructive proof for the theorem since by the algorithms, overlapping trigger paths are separated, effectively making them ordinary acyclic graphs, and the topological sort is performed on the graphs. □

3.1.4 Parallel Rule Executions

The execution graph naturally allows parallel execution of rules. In the extended form, such as Figure 3.3(b), all rules with indegree 0 can be launched in parallel for execution. Since there should be no dependency edges between nodes with indegree 0 in an execution graph,[2] relative execution order of those independent rules does not

[2] Note that an execution graph reduces its size as rules are executed.

Given EG:

Schedule()
{
while $(EG \neq \emptyset)$ do
    {
    choose a node r_i with indegree 0;
    execute r_i;
    M[i]--; // decrease rule_count of node r_i
    for all r_j such that $\exists(r_i \xrightarrow{T} r_j) \in EG$ do
        if (r_j was not triggered by execution of r_i) then
            call DFS_Dec_M(r_j);
    if (M[i] $= 0$) then
        delete node r_i and edges $(r_i \xrightarrow{T} r_k)$ and $(r_i \xrightarrow{D} r_l)$,
            for any k and l, from EG;
    }
}

DFS_Dec_M(r_j)
{
M[j]--;
for all r_k such that $\exists(r_j \xrightarrow{T} r_k)$ do
    call DFS_Dec_M(r_k);
if (M[j] $= 0$) then
    delete node r_j and edges $(r_j \xrightarrow{T} r_l)$ and $(r_j \xrightarrow{D} r_m)$,
        for any l and m, from EG;
// don't need to delete dependency edges incoming to r_j!
}

Figure 3.6. Algorithm – Schedule()

affect the final outcome. Note also that multiple instances of the same rule can be scheduled for execution at the same time.

In an execution graph with rule_counts, which we use for our work, all rules with indegree 0 are scheduled simultaneously as many times as the rule_counts associated with the rules. In Figure 3.4, for instance, after scheduling and executing each one instance of r_1 and r_3 in parallel, two instances of r_2 can be scheduled for execution since rule_count of r_2 is 2. In order to implement the parallel rule executions, we have to make some changes to the algorithm of Figure 3.6 which we will not elaborate in this work. Whenever execution of one instance of a rule is completed, the associated rule_count need be decreased by 1 and its child rules have to be checked to see whether they are triggered or not by the parent rule. If some are not triggered, *DFS_Dec_M()* should be called to recursively decrease rule_counts along trigger paths. Since the rule_count array *M[]* and execution graph are shared data structures, some locking mechanism need be used to avoid update anomalies within the data structures.

One important measure in parallel processing is the degree of parallelism. In the active rule system, the maximum parallelism is bounded by dependencies between rules in the system rule set. For instance, if all the rules are independent of each other, ideally all triggered rules can be executed in parallel. As dependencies between rules increase, the degree of parallelism would decrease. However, other components too can restrict parallelism. As discussed in Section 2.2, improper priority specification and rule execution model may execute a given rule set serially which could be executed in parallel. Specifically in our work, two components can hamper parallelism, all resulted from static analysis. First, precision of dependency analysis between two rules can affect parallelism. Even though there is data dependency between two rules, they can be commutative in reality. Being unable to detect such a hidden commutativity results in a false dependency edge in an execution graph, likely costing

parallelism. Second, precision of trigger relationship analysis can similarly affect parallelism. If we know for sure that one rule triggers another rule, the trigger edge between the two rules can be deleted after all rule_count values are computed. This way, the two rules can be scheduled in parallel if there is no other path connecting them in the resultant execution graph. Using static analysis, we cannot completely avoid uncertain trigger edges, and presence of the uncertain trigger edges can cost parallelism. However, ignoring the loss caused by imprecision of static analysis, the strict order-preserving rule executions exploit the maximum parallelism existing in a given system rule set. We state it in Theorem 2.

Theorem 2 *Using the strict order-preserving rule executions, the active rule execution model achieve the maximum parallelism within limitations of static trigger and dependency analysis.*

Proof of Theorem 2 Given any acyclic extended execution graph, there are two kinds of edges; trigger edges and dependency edges. We first assume that no trigger edges can be removed, that is, they are all uncertain trigger edges. Now suppose there are superfluous dependency edges in the execution graph whose absence does not affect the final database state. (Therefore we can remove them safely to increase parallelism.) There can be only two types of dependency edges in an acyclic execution graph. Given any dependency edge $(r_i \stackrel{D}{\rightarrow} r_j)$, r_i is either a proper ancestor of r_j in a trigger path containing both r_i and r_j or not an ancestor of r_j in any trigger path. In the first case, even though the dependency edge is redundant in terms of representation, removal of that edge does not allow r_j to execute before r_i since r_j is yet to be triggered by r_i or its descendant. Thus, removal of the first type of dependency edges has no effect of increasing parallelism. In the second case, if $(r_i \stackrel{D}{\rightarrow} r_j)$ is the only dependency path that can be interconnected by trigger edges and dependency edges to connect r_i to r_j, obviously this cannot be removed at any rate. If

there exist other dependency paths connecting r_i to r_j whose lengths are longer than $(r_i \xrightarrow{D} r_j)$ (of course, if such paths exist, they should be longer than one-edge path $(r_i \xrightarrow{D} r_j)$), $(r_i \xrightarrow{D} r_j)$ is redundant, but again, removal of the dependency edge does not allow r_j to execute before r_i. By applying this argument to all dependency edges present in the extended execution graph, we can see that dependency edges are either necessary or redundant, but removal of redundant edges does not increase parallelism. Since the execution graph with rule_counts are equivalent to the extended execution graph under the strict order-preserving rule executions, we can conclude that the rule scheduler exploit the maximum parallelism inherent in the system rule set. □

3.2 Alternative Policies for Handling Overlapping Trigger Paths

3.2.1 Serial Trigger-Path Executions

In order to handle overlapping trigger paths, there may be other approaches than the strict order-preserving rule executions. One obvious approach among them is *serial trigger-path execution* in which all rules in an overlapping trigger path execute before any other rules in other overlapping trigger paths. In other words, when trigger paths overlap, rules in those overlapping trigger paths execute in trigger-path by trigger-path fashion. For instance, in Figure 3.1, $\langle r_i \cdot r_j \cdot \bar{r_k} \cdot \bar{r_l} \cdot r_j \cdot \bar{r_k} \cdot \bar{r_l} \rangle$ is a serial trigger-path execution.

Although the serial trigger-path execution could appear more intuitive, unfortunately it brings forward the old problem again. Different serial trigger-path executions may result in different final database states when there exist dependencies between the overlapping trigger paths. Therefore the user has to choose one serial order over the conflicting overlapping trigger paths to obtain a unique final database state. However, choosing a serial order in advance (i.e., at compile time) is not always possible, since, as discussed in Section 2.2, multiple instances of the same rule may be in UTRS and one cannot predict them before the rules execute. To reiterate

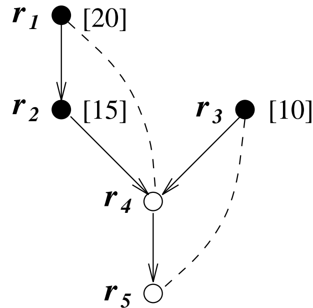

Figure 3.7. A priority graph with absolute priorities

that, let's assume that A, B, and C are three overlapping trigger paths rooted by rules a, b, and c, respectively, and the trigger-paths are all conflicting each other. Then, if each one instance of a, b, and c is in UTRS, any of six permutations (3!) made of A, B, and C (i.e., serial trigger-path executions) may produce a different result. However, if rule a is triggered twice while the other rules remain as before, four trigger paths A, A', B, C will be instantiated and the permutations increase to 24 (4!). As more instances of a, b, or c are added to UTRS, the permutation will increase exponentially.

From the observation above, it is apparent that we again have to settle for a less general scheme, which will be discussed below, than the full-fledged serial trigger-path execution. Figure 3.7 shows a priority graph that is slightly different from Figure 3.3(a). The new graph has dependency edges between r_1 and r_4, and between r_3 and r_5 as before. Also, it uses absolute priorities (numbers in brackets) attached to rules rather than directed dependency edges. Using this priority graph, the rule scheduler will be able to schedule a serial trigger-path execution by executing rules in a trigger-path whose root rule's priority is the highest. For instance, if rules r_1, r_2, and r_3 are each triggered once by a user transaction, given the priorities, the serial trigger-path execution being scheduled will be $\langle \bar{r_1} \cdot r_2 \cdot \bar{r_4} \cdot \bar{r_5} \cdot r_2 \cdot \bar{r_4} \cdot \bar{r_5} \cdot \bar{r_3} \cdot \bar{r_4} \cdot \bar{r_5} \rangle$. Now if r_2 is triggered twice and r_1 and r_2 once, then since priority of r_2's instances is

lower than r_1 and higher than r_3, the resultant serial trigger-path execution will be $\langle \bar{r_1} \cdot r_2 \cdot \bar{r_4} \cdot \bar{r_5} \cdot r_2 \cdot \bar{r_4} \cdot \bar{r_5} \cdot r_2 \cdot \bar{r_4} \cdot \bar{r_5} \cdot \bar{r_3} \cdot \bar{r_4} \cdot \bar{r_5} \rangle$.

Althouh we will not elaborate in this work, it will be worthwhile to investigate more efficient and convenient ways of specifying serial execution orders, if the serial trigger-path execution is to be pursued for implementation. Using absolute priorities could be a hassle as pointed out in Section 2.2. On the other hand, sacrificing generality in serial order specification might do more good than harm. For example, it can be assumed that in a trigger path, ancestors have priority over their descendents when they are in UTRS (or vice versa). If such an assumption is made, the explicit priority specification between r_1 and r_2 in Figure 3.7 can be omitted. It should be noted that in order to schedule serial trigger-path executions, the rule scheduler described in the previous section need be modified appropriately to deal with the added absolute priorities.

3.2.2 Serializable Trigger-Path Executions

A more interesting scheduling than the serial trigger-path execution would be *serializable trigger-path execution* in which rule execution sequence may not be a serial trigger-path execution but the final result is equivalent to that of *a* serial trigger-path execution. In Figure 3.7, for example, one can see that a rule execution sequence $\langle r_i \cdot r_j \cdot r_j \cdot \bar{r_k} \cdot \bar{r_l} \cdot \bar{r_k} \cdot \bar{r_l} \rangle$ is equivalent to the serial trigger-path execution $\langle r_i \cdot r_j \cdot \bar{r_k} \cdot \bar{r_l} \cdot r_j \cdot \bar{r_k} \cdot \bar{r_l} \rangle$. Therefore, the former is a serializable trigger-path execution.

Once a serial execution order of overlapping trigger paths is set, it can be readily translated to a serializable trigger-path execution using extended execution graph. Figure 3.8 illustrates the first step to translate a serial trigger-path execution, shown in Figure 3.7 assuming r_2 is triggered twice, to an equivalent serializable trigger-path execution. In Figure 3.8, dashed lines, presently undirected, represent dependencies between rules (only inter-trigger-path dependencies are shown), and directed dotted

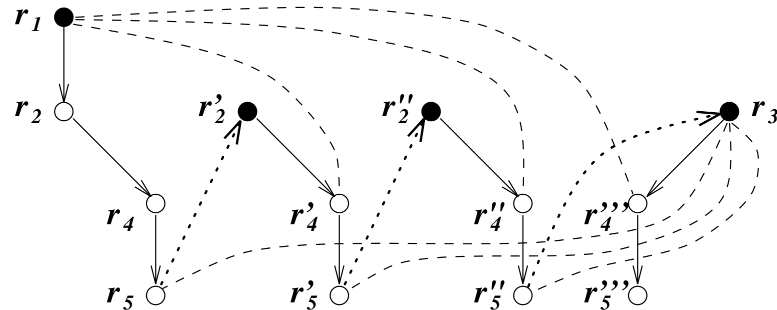

Figure 3.8. Translation to serializable trigger-path execution – Step 1

lines connecting the last rule in a trigger path to the first rule in the next trigger path (between r_5 and r'_2, for example) impose the serial order specified in Figure 3.7. Ignoring the undirected dependency edges and regarding the newly added dotted arrows as only dependency edges, our rule scheduler will schedule the original *serial* trigger-path execution. Then, as the second step of translation, directions of dependency edges are set by following trigger paths bridged by the dotted arrows so that the directions are consistent with the serial order. After that, the dotted arrows are deleted. Figure 3.9 shows the result of the second step translation. Now if the translated execution graph is fed to the rule scheduler, an equivalent serializable trigger-path execution will be generated. Note that there may be multiple equivalent serializable trigger-path executions and they are those that enable parallel rule executions. Maximum parallelism of Theorem 2 still holds for the translated execution graph.

3.2.3 Comparisons with Strict Order-Preserving Execution

Collating the serializable (or serial) trigger-path execution with the strict order-preserving execution raises an interesting point. Although choosing between them would remain as largely subjective a matter, entrepreneurial policies and applications' semantics could favor one over the other. Suppose rule r_i triggers rule r_j and there is a dependency between them. When multiple instances of r_i are triggered

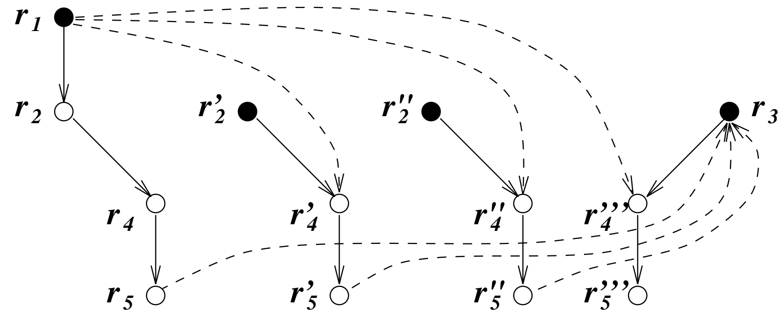

Figure 3.9. Translation to serializable trigger-path execution – Step 2

by a user transaction, in the traditional transaction processing environment, the serializable trigger-path execution would be favored since, regarding a trigger path as a subtransaction, it appears to fit well into what already exist in that environment. On the other hand, when r_i stands for an absolute raise to an employee's salary (say, $1000) and r_j for a relative raise (say, 5%), then the strict order-preserving execution would make more sense, if the company has a policy that applies all applicable absolute raises before relative raises. Considering the serializability is often used as a correctness criterion for executions of subtransactions in the nested transaction which, in turn, has been proposed to be used for rule executions in some active databases [6, 12], this example illustrates that the serializability may not be an appropriate choice for some cases even if it is strengthened to yield confluent rule executions.

As for implementation, the serializable trigger-path execution is less favorable in our current framework. Compared to the strict order-preserving execution, it requires a different priority specification scheme than the simple priority graph. However, once transformed properly to the form of Figure 3.9, the serializable trigger-path execution is on the par with the strict order-preserving execution in terms of exploiting maximum parallelism in rule executions.

3.3 Discussion and Conclusions

In this work we have proposed a new active-rule execution model along with priority specification schemes to achieve confluent rule executions in active databases. As other rule execution models, we employ prioritization to resolve conflicts between rules. Ideally, by prioritizing executions of conflicting rules whose different relative execution orders can yield different database states, one can achieve confluent rule executions. It is necessary, however, to prioritize as few rules as possible since prioritizing many rules in a meaningful way itself could be a challenge and the more rules prioritized, the less rules being able to execute in parallel. Prioritizing conflicting rules only, on the other hand, may call for incorrect results as executions of seemingly independent rules can trigger and execute conflicting rules in the wrong way. Also, when rules in the same trigger path are triggered resulting in overlapping trigger paths, more problems can be brought up. Unlike previous rule execution models, our model uses a rule scheduler based on the topological sort to respect all the specified priorities if applicable. This way, rules being triggered and executed from a user transaction can follow the execution sequence imposed by the priority specification to make their execution confluent. We also have proposed the strict order-preserving rule execution to deal with overlapping trigger paths. In the strict order-preserving rule execution, when a part of or the whole trigger path is multiply executed and there are priorities between rules in the trigger path, the rules are executed in such a way that no rule appears before rules with higher priorities if any. It has been shown that our rule execution model can exploit maximum parallelism in rule execution. Lastly, we have discussed other possible ways of handling overlapping trigger paths, i.e., the serial trigger-path execution and serializable trigger-path execution.

There are other related issues not pursued in our current work. One of such issues is precision of data dependency (or dependency in general). Our definition in

Section 2.3.2 may be too coarse as some rules might be commutative despite presence of the defined dependencies. If an active rule language has a definite form as SQL, the definition of dependency and its detection may be tightened by analyzing static expressions in rule definition as done in Baralis and Widom [7]. In Sentinel, a rule has a very general form since the condition and action parts can be arbitrary functions. In such a system, even detecting dependency with a margin of imprecision can be challenging. However, we know empirically that in general only a few stored data items are referenced in those functions. The rule designer should be able to readily recognize the referenced data items and classify them into read-set and write-set. Once the read-set and write-set are obtained, the dependency graph can be drawn as usual.

The problem of confluent executions can be further complicated in those active databases such as Sentinel [13] and HiPAC [12] where *coupling modes* between event and condition and between condition and action are defined. In Sentinel and HiPAC three coupling modes are defined: *immediate*, *deferred*, and *detached* modes. For coupling modes between event and condition, the immediate mode prescribes that condition be tested immediately after a relevant event is detected. In deferred mode, condition is tested after all user-defined operations (except commit/abort) in the current transaction are performed. In detached mode, condition is tested in a different transaction. The semantics of these modes holds for the coupling modes between condition and action as well.

Our work described in this thesis assumes the immediate mode between event and condition and between condition and action as well. However, considering that in the deferred mode, tests (or actions) are carried out in the same order as the occurrence order of events that triggered the rules, our model works well for deferred-deferred couplings between event and condition and between condition and action.

Interestingly, in a situation where event-condition coupling is the immediate mode and condition-action coupling is the deferred mode, our model is still applicable. But this combination of coupling modes precludes the possibility of one rule's untriggering the others since all tests are completed before any action is performed. On the other hand, the detached mode doesn't suit with our model and other models as well that deal only with conflicts within a transaction boundary. By detaching the execution of action part from the current transaction, even execution of only one rule can result in different final database states depending on interaction between the detached transaction and the parent transaction. And those interactions are governed by transaction model the system employs. Unless a rule execution model is geared with the transaction model, there is no room to control the inter-transaction interactions to make rule executions confluent.

CHAPTER 4
AGGREGATE CACHE

4.1 Motivation

Complex aggregate expressions are frequently computed in data warehouses to support decision making and for statistical data analysis. These computations are expensive to perform as they require at least one full scan of usually huge (or even distributed) base databases (or operational databases). Hence, it is vital for the data warehouse applications to have a means of reducing the computation overhead. An approach used to address this problem is to *materialize* views defined in the data warehouse, assuming that both the underlying base databases and the data warehouse use the relational data model. There already exists a body of literature on view materialization in the context of relational databases or deductive databases [8, 22, 25, 27]. Recently, the same issue has been reexamined by several researchers [23, 24, 40] from the data warehouse viewpoint. However, bulk of the literature deals with SPJ (Select-Project-Join) views. Quite a few papers mention materialization of aggregates as a means of performance improvement, but no in-depth study has been reported except [32]. Using view materialization, some of SPJ views defined in the data warehouse schema are chosen to be materialized. Then, when base databases change, materialized views that are affected by the change are updated in an incremental way if possible; otherwise the affected views are rematerialized. Realistically, however, application of traditional view materialization techniques seems inappropriate for the data warehouse environment for the following two reasons:

1. Unlike views, aggregates are usually not predefined and it is hard to predict what aggregates on what data items are computed, since aggregates tend to be used in ad hoc queries.

2. Due to the natural process of data analysis or decision making, many aggregates are not likely to be used again (or rarely used) after some period of heavy usage.

Note that the second reason above implies the presence of *temporal locality* property of aggregates usage in data warehouse applications. This is one good reason to introduce the *cache* mechanism for handling aggregates in data warehouses. In addition, by using a cache, an aggregate is cached the first time when it is actually used. That is, the user is not required to declare in advance what aggregates would be used. Therefore this is also a good way to deal with ad hoc queries containing aggregates.

In addition to the temporal locality, the *spatial locality* is also observed in aggregate usage in data warehouse applications. That is, when an aggregate is used in a query, other aggregates closely related to this aggregate in terms of data modeling perspectives are also expected to be queried sooner or later. For example, if average salary of all employees in a department has been queried, it is more likely that average age of the same group of employees is queried in a short period of time than total order placed today. Unfortunately, this sort of spatial locality is too loose and abstract to be used to enhence efficience of cache. Therefore we will not consider the spatial locality in our work.

In this work we propose an *aggregate cache*. As Figure 4.1 shows, the aggregate cache can be added to a data warehouse as a plug-in. The aggregate cache uses part of the data warehouse (i.e., disks) as a cache. It interacts with query processor of the data warehouse manager (therefore, the query processor need be modified to some extent) to expedite computation of aggregates. It also interacts with the integrator,

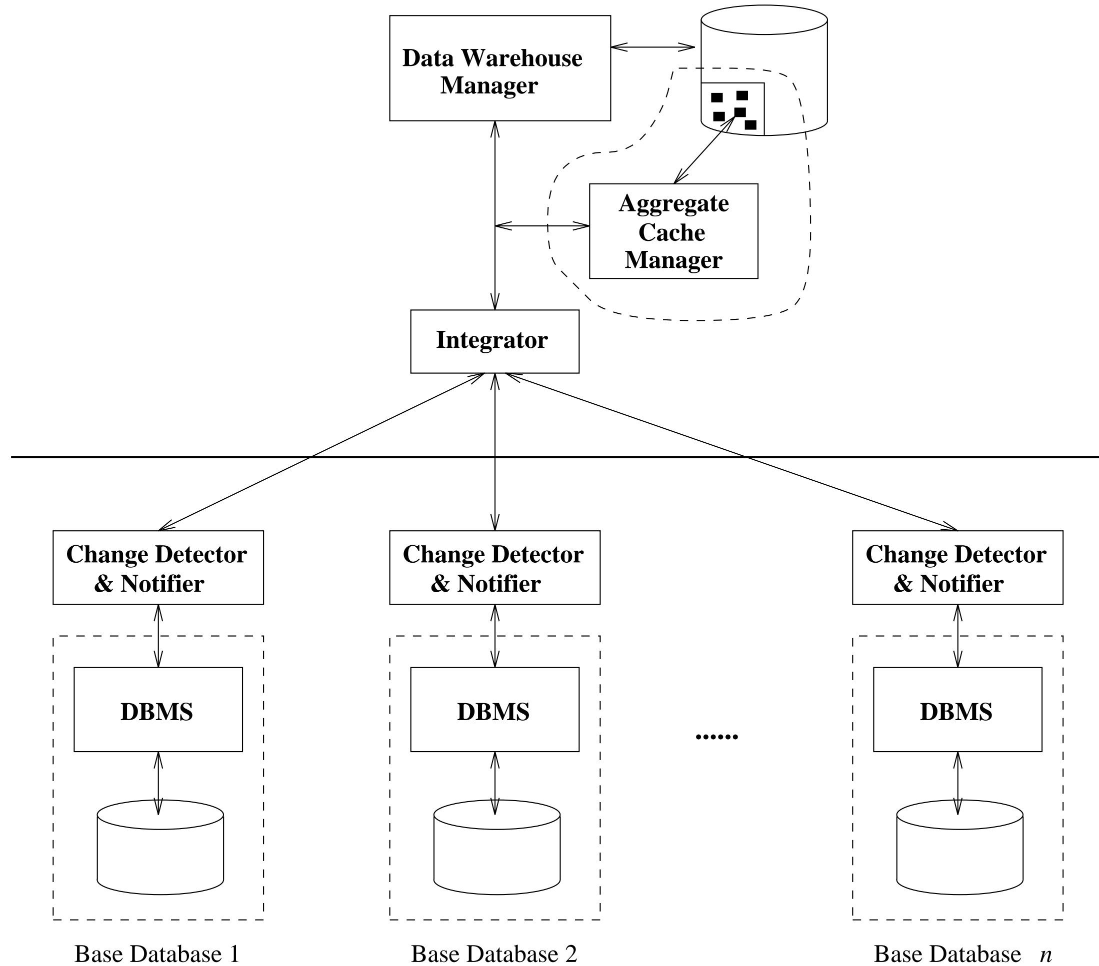

Figure 4.1. A Data Warehouse and Aggregate Cache

which intercepts change notifications from base databases that are relevant to cached aggregates, and updates affected cached aggregates appropriately.

When an aggregate in a query (of the data warehouse user) is processed, the query processor of the data warehouse manager first consults the aggregate cache manager. If the aggregate is found in the cache, the stored value is used for processing the query. Otherwise, the aggregate needs to be re-computed from data in base databases or from data in the data warehouse if an appropriate (presumably, summarized) copy is present there. In any case, the computed aggregate is cached for later use, unless otherwise directed.

For the aggregate cache proposed in this work, not all aggregates need be cached. If the user knows that some of the aggregates will not be used again, the user should be able to mark such aggregates in queries so that they are not cached by the aggregate cache. On the other hand, if a certain set of aggregates is known to be used frequently, these aggregates can be pre-computed and installed in the cache and will not be replaced until explicitly requested to do so. Between these two extremes, all ordinary aggregates are cached when they are first referenced in queries and replaced out later by a cache replacement policy. By taking this approach, as far as aggregates are concerned, the aggregate cache as a whole behaves as an adaptive schema for a data warehouse that dynamically adapts itself to uncertain and varying user requirements.

4.2 Updating Cached Aggregates

As base databases change, there should be some way of reflecting those changes to aggregates cached in the data warehouse, if the changes are relevant. Presuming affected aggregates are known, one way is to invalidate the affected aggregates. This method was addressed by Sellis [32] in the context of maintaining derived data in relational databases. However, in data warehouse environments, where changes to base databases are expected to be frequent (one of the very reasons why the data warehouse was introduced), simply invalidating affected aggregates will not be a good approach as cached aggregates would not have much chance to be reused. Therefore, our natural choice is updating affected aggregates as necessary.

Again assuming affected aggregates are known when base databases change, there may be several ways of updating the aggregates as outlined below:

Rematerialization:

Affected aggregates are recomputed by rescanning base databases.

Periodic Update:

Changes to base databases are kept in respective base databases and periodically the changes are propagated to the data warehouse. The data warehouse, then, collects the changes and *incrementally* updates affected aggregates.

Eager Update:

Every time a change is made to a base database, if the change is relevant to any cached aggregates, it is propagated to the data warehouse so that the data warehouse can incrementally update the affected aggregates.

On-demand Update:

Changes to base databases are accumulated either on the sides of base databases or on the side of data warehouse, forming *delta files*. When a cached aggregate is accessed by a query, relevant delta files are collected and the aggregate is updated using the delta files.

The rematerialization approach is the most expensive. In the aggregate cache, rematerializing affected aggregates prior to their usage makes little sense unless the system load is particularly light at that time. This method, however, can be combined with an incremental method to handle a few cases where the incremental method cannot be applied.

A naive implementation of the periodic update could be troublesome. If the frequency of update is too low, values of cached aggregates will be outdated. If the frequency is too high, on the other hand, it will degrade the performance of system. Rather than independently used, the periodic update can be geared with the on-demand update as it will be explained later.

The eager update could appear to be the best way to deliver the up-to-date values of cached aggregates quickly. However, unless the system's timeliness requirement is

so stringent, this method is considered to be an overkill since generally one does not know when cached aggregates are to be used and whether or not they would ever be used again before they are replaced out. Furthermore, there is no guarantee that the eager update will always outperform other methods in the timeliness measure since it could take up too much of system resources if base databases change often and there are many cached aggregates to update. In fact, a simulation result reported by Adelberg et al. [1], in which various recomputation methods for maintaining derived data were compared, shows that the eager update is inferior to the on-demand update in terms of query response time. Although the simulation setting does not completely match the situation of aggregate maintenance in a data warehouse environment, we think we can safely extrapolate this result to draw a similar result in the aggregate cache.[1]

The on-demand update is considered the best fit for the aggregate cache since the frequency with which the base databases change is expected to be much higher than the access frequency of aggregates in the data warehouse. As mentioned earlier, the delta files can be kept either in base databases or in the data warehouse. Keeping delta files in data warehouse is, in effect, more like the eager update since whenever a related base database changes, the change need be transferred to the data warehouse although, unlike the eager update, recalculations of affected cached aggregates happen only when they are referenced. On the other hand, keeping delta files in each base database will cause some delay when an affected cached aggregate is referenced. If

[1] We expect that the gap between the eager update and the on-demand update will widen in the aggregate cache, since the simulation by Adelberg et al. [1] does not consider the overhead of change propagation between a base database and the data warehouse, which is likely to be substantial. Extensive change propagation in the eager update will deteriorate the performance of the eager update further. Moreover, we can regard an aggregate as a derived data with a huge "fan-in," which is used in the simulation to describe the number of source data for a derived datum, since an aggregate need be recomputed whenever a row in a referenced relation is modified. According to the simulation result, the gap between the eager update and the on-demand update widens as the fan-in increases.

the data warehouse and base databases are connected through a network, network connection and data transfer time will dominate the delay. In our work, by default we keep delta files in base databases, but if a base database has difficulty in keeping them and the network overhead between the base database and the data warehouse is not severe, delta files for that base database should be able to be kept in the data warehouse.

An important premise for using the on-demand update (the eager update and periodic update as well) is that aggregates can be computed incrementally. Although a large class of aggregates can be computed incrementally, there are some aggregates which cannot be computed incrementally. Such aggregates include order-related ones such as *min*, *max*, and *median* [26, 27]. When an aggregate cannot be computed incrementally, the aggregate is rematerialized (i.e., recomputed) *on demand*.

One problem with the on-demand update is that for a cached aggregate, if related base databases change frequently and the aggregate is not referenced for a long time, delta files can grow too big in base databases. This problem can be handled in several ways:

1. The data warehouse *periodically* polls base databases on behalf of cached aggregates that are not referenced for a certain period of time. Base databases, then, repond by sending delta files to the data warehouse. This is the periodic update.

2. Each base database monitors sizes of delta files. If size of a delta file grows over a limit, the base database notifies the data warehouse and sends the oversized delta file to it. This is an aperiodic (or asynchronous) counterpart of the periodic update.

3. Instead of keeping delta files, base databases keep *numeric delta*'s of affected aggregates and the numeric values are sent when aggregates are referenced in the

data warehouse. As a simplistic example, when aggregate *count(*)* on a relation in a base database is cached, the numeric delta in this case is the difference in the numbers of tuples inserted into and deleted from that relation. When the aggregate is referenced, the numeric delta is sent to the data warehouse, instead of delta file containing inserted and deleted tuples.

4. The overgrown delta files are discarded and related cached aggregates are replaced out.

Clearly, the aperiodic update (the second one above) is better than the periodic update since in the aperiodic update, transfer of delta files takes place only when necessary.

The numeric delta method (the third one) has an effect of distributing burden of the data warehouse to base databases. If base databases can sustain the newly imposed load, the performance of data warehouse should improve since network traffic can be reduced substantially and incremental updates in the aggregate cache become simpler. A drawback of this method is the added complexity, especially on the sides of base databases. Base databases now have to know what aggregates are being cached and all the information needed to perform partial computations of affected aggregates.

Discarding overgrown delta files is closely coupled with the cache replacement policy. If the policy is LRU (Least-Recently-Used) or its variation, cached aggregates not referenced for a long period of time can be replaced out or just flushed. Therefore, it is justifiable to discard overgrown delta files if related cached aggregates are not used for a long time. Of course, if delta files are to be discarded, related cached aggregates should also be deleted.

To sum up, in our aggregate cache, we use both aperiodic update and numeric delta in conjunction with the on-demand update. The aperiodic update is used when

a base database is unable to implement the numeric delta method due to its limited functionality or when an aggregate does not allow the numeric delta method.

4.3 Incremental Update of Aggregates

As mentioned in the previous section, the incremental update is a premise for the on-demand update that is adopted in the aggregates cache. In our current work, an *aggregate* is defined in a general way to be a function that takes as input one or more nonempty sets of objects, called *aggregate input sets* and zero or more scalar variables and returns as output one scalar value. Elements in an aggregate input set are denoted by a variable called *aggregate variable*. An aggregate input set corresponds to a group of values of an attribute in a relational table.

4.3.1 Syntactic Conventions

We present syntactic conventions used in subsequent sections.

- Aggregate input sets are denoted by capital letters such as X, Y, and Z. If size of a set X (i.e., cardinality) is of significance and the size is n, a positive integer, the set is denoted by X_n.

- Aggregate variable of an aggregate input set X is denoted by the lower-case letter, x, and it represents an element in X. For X_n, elements in the set are distinguished by attaching a distinct subscript to x as $X_n = \{x_1, x_2, \cdots, x_i, \cdots, x_n\}$.

- Aggregates are denoted by capital script letters such as $\mathcal{F}$ and $\mathcal{H}$. An aggregate $\mathcal{F}$ with its arguments can be denoted as follows:

  $$\mathcal{F}(X_n, Y_m, \cdots, Z_o, \alpha, \beta, \cdots, \gamma).$$

  where $X_n, Y_m, \cdots, Z_o$ are aggregate input sets and $\alpha, \beta, \cdots, \gamma$ are zero or more independent scalar variables.

4.3.2 Incrementally Updatable Aggregates

Informally, incremental update of an aggregate means that when a new element is added to the aggregate input set (assuming the aggregate has only one input set) or an old element is deleted from the aggregate input set, the new value of the aggregate is computed from the added (or deleted) element and the current value of the aggregate stored in the system. If an aggregate has multiple aggregate input sets, let's assume for the time being that all the aggregate input sets are of the same size and insertions or deletions take place such a way that all the sets remain in the same size. Many aggregates used in statistical analysis fall into this category. In order to define incremental update precisely, we first define positive delta and negative delta below.

Let's assume an aggregate $\mathcal{F}(X_n, Y_n, \cdots, Z_n, \alpha, \beta, \cdots, \gamma)$. For convenience, let Ω_{n-1} and Ω_n be two sets containing parameters for $\mathcal{F}$ as follows:

$$\Omega_{n-1} = \{X_{n-1}, Y_{n-1}, \cdots, Z_{n-1}, \alpha, \beta, \cdots, \gamma\},$$
$$\Omega_n = \{X_n, Y_n, \cdots, Z_n, \alpha, \beta, \cdots, \gamma\}.$$

Now, suppose that for the aggregate $\mathcal{F}$, the current aggregate input sets are $X_{n-1}, Y_{n-1}, \cdots, Z_{n-1}$, and $x_n, y_n, \cdots, z_n$ are *inserted* elements to their respective input sets (thereby making them $X_n, Y_n, \cdots, Z_n$, respectively).

Positive delta, Δ_n is defined such that

$$\Delta_n = \mathcal{F}(\Omega_n) - \mathcal{F}(\Omega_{n-1}). \tag{4.1}$$

Aggregate $\mathcal{F}$ is incrementally updatable on the inserted elements into the aggregate input sets, if there exists an algebraic function g such that

$$\begin{aligned} \Delta_n \quad = \quad & g(\mathcal{F}_1(\Omega_{n-1}), \mathcal{F}_2(\Omega_{n-1}), \cdots, \mathcal{F}_i(\Omega_{n-1}), \\ & \mathcal{H}_1(\Omega_n), \mathcal{H}_2(\Omega_n), \cdots, \mathcal{H}_j(\Omega_n),\ x_n, y_n, \cdots, z_n) \end{aligned} \tag{4.2}$$

where all $\mathcal{F}_k$'s $(1 \leq k \leq i)$ and $\mathcal{H}_l$'s $(1 \leq l \leq j)$ are aggregates whose values are already known or incrementally updatable and none of $\mathcal{H}_l$'s is $\mathcal{F}$.

For the same aggregate $\mathcal{F}$, suppose that the current aggregate input sets are $X_n, Y_n, \cdots, Z_n$ and $x'_n, y'_n, \cdots, z'_n$ are *deleted* elements from their respective input sets (thereby making them $X_{n-1}, Y_{n-1}, \cdots, Z_{n-1}$).

Negative delta, $\overline{\Delta}_{n-1}$ is defined such that

$$\overline{\Delta}_{n-1} = \mathcal{F}(\Omega_{n-1}) - \mathcal{F}(\Omega_n). \tag{4.3}$$

Aggregate $\mathcal{F}$ is incrementally updatable on the deleted elements from the aggregate input sets, if there exists an algebraic function $\overline{g}$ such that

$$\begin{aligned} \overline{\Delta}_{n-1} \quad &= \quad \overline{g}(\mathcal{F}_1(\Omega_n), \mathcal{F}_2(\Omega_n), \cdots, \mathcal{F}_i(\Omega_n), \\ &\qquad \mathcal{H}_1(\Omega_{n-1}), \mathcal{H}_2(\Omega_{n-1}), \cdots, \mathcal{H}_j(\Omega_{n-1}),\ x'_n, y'_n, \cdots, z'_n) \end{aligned} \tag{4.4}$$

where all $\mathcal{F}_k$'s ($1 \leq k \leq i$) and $\mathcal{H}_l$'s ($1 \leq l \leq j$) are aggregates whose values are already known or incrementally updatable and none of $\mathcal{H}_l$'s is $\mathcal{F}$.

An aggregate $\mathcal{F}$ is defined to be *incrementally updatable*, if $\mathcal{F}$ has both positive delta Δ_n (equation 4.1) and negative $\overline{\Delta}_{n-1}$ (equation 4.3) that are computable for all $n > 1$.

Once Δ_n is computed, $\mathcal{F}(\Omega_n)$ in equation 4.1 can be obtained by adding Δ_n to the known value of $\mathcal{F}(\Omega_{n-1})$. Likewise, from $\overline{\Delta}_{n-1}$, $\mathcal{F}(\Omega_{n-1})$ in equation 4.3 is obtained by adding $\overline{\Delta}_{n-1}$ to the known value of $\mathcal{F}(\Omega_n)$.

Now, if an aggregate is sensitive to every insertion into (or a deletion from) any one aggregate input set, the definition of positive delta and negative delta can be extended in a straightforward way so that the aggregate can be incrementally updated whenever an insertion (or a deletion) is made. We would not elaborate the extension in this work.

4.3.3 Algebraic Aggregates and Non-Algebraic Aggregates

Aggregates are first classified into algebraic ones and non-algebraic ones. An *algebraic aggregate* is an aggregate that takes as input only aggregate input sets of

real numbers and independent scalar variables of the real number and returns as output one real number and consists of only algebraic operations defined over the real number.[2] A *non-algebraic aggregate* is an aggregate that its domain or range is non-numeric or uses non-algebraic operations.

<u>*Example 4.3.1*</u> Here are some examples of algebraic aggregates.

$$count(X_n) = \sum_{i=1}^{n} 1$$

$$sum(X_n) = \sum_{i=1}^{n} x_i$$

$$average(X_n) = \frac{\sum_{i=1}^{n} x_i}{\sum_{i=1}^{n} 1} = \frac{sum(X_n)}{count(X_n)}$$

$$S_{xy}(X_n, Y_n) = \sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y}) = \sum_{i=1}^{n} x_i y_i - \frac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n}$$

$S_{xy}(X_n, Y_n)$ is sum of products of distances of x and y from their means, $\overline{x}$ and $\overline{y}$ (i.e., $average(X_n)$ and $average(Y_n)$), and is used to compute simple linear regression.

◁

<u>*Example 4.3.2*</u> Non-algebraic aggregates include $max(X_n)$, $min(X_n)$, $median(X_n)$, and $r\text{-}th_percentile(X_n, r)$ which is the value such that r percent of x's in X_n fall at or are below that value. These aggregates all involve procedural operations, thus not algebraic aggregates. ◁

<u>*Example 4.3.3*</u> For some simple algebraic aggregates, positive delta (Δ) and negative delta ($\overline{\Delta}$) defined in Section 4.3.2 can be directly obtained by algebraic manipulations. Below, we show how positive delta and negative delta of $average(X_n)$ are derived.

$$\Delta_n = average(X_n) - average(X_{n-1})$$

[2]Note that the real number subsumes the integer. Therefore, aggregates taking integer sets and return an integer or real number, such as *count(*)*, can be included by the definition.

$$
\begin{aligned}
&= \frac{\sum_{i=1}^{n} x_i}{n} - \frac{\sum_{i=1}^{n-1} x_i}{n-1} \\
&= \frac{(n-1)\sum_{i=1}^{n} x_i - n\sum_{i=1}^{n-1} x_i}{n(n-1)} \\
&= \frac{n\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} x_i - n\sum_{i=1}^{n-1} x_i}{n(n-1)} \\
&= \frac{n(\sum_{i=1}^{n} x_i - \sum_{i=1}^{n-1} x_i) - \sum_{i=1}^{n} x_i}{n(n-1)} \\
&= \frac{nx_n - \sum_{i=1}^{n} x_i}{n(n-1)} \\
&= \frac{nx_n - \sum_{i=1}^{n-1} x_i - x_n}{n(n-1)} \\
&= \frac{(n-1)x_n - \sum_{i=1}^{n-1} x_i}{n(n-1)} \\
&= \frac{(n-1)x_n}{n(n-1)} - \frac{\sum_{i=1}^{n-1} x_i}{n(n-1)} \\
&= \frac{x_n}{n} - \frac{average(X_{n-1})}{n} \\
&= \frac{x_n - average(X_{n-1})}{n} \\
&= \frac{x_n - average(X_{n-1})}{count(X_n)}
\end{aligned}
$$

$$
\begin{aligned}
\overline{\Delta}_{n-1} &= average(X_{n-1}) - average(X_n) \\
&= \frac{\sum_{i=1}^{n-1} x_i}{n-1} - \frac{\sum_{i=1}^{n} x_i}{n} \\
&= \frac{n\sum_{i=1}^{n-1} x_i - (n-1)\sum_{i=1}^{n} x_i}{(n-1)n} \\
&= \frac{n\sum_{i=1}^{n-1} x_i - n\sum_{i=1}^{n} x_i + \sum_{i=1}^{n} x_i}{(n-1)n} \\
&= \frac{-n(\sum_{i=1}^{n} x_i - \sum_{i=1}^{n-1} x_i) + \sum_{i=1}^{n} x_i}{(n-1)n} \\
&= \frac{-nx_n + \sum_{i=1}^{n} x_i}{(n-1)n} \\
&= \frac{-nx_n}{(n-1)n} + \frac{\sum_{i=1}^{n} x_i}{(n-1)n} \\
&= \frac{-x_n}{n-1} + \frac{average(X_n)}{n-1}
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{average(X_n) - x_n}{n-1} \\
&= \frac{average(X_n) - x_n}{count(X_{n-1})}
\end{aligned}
$$

Δ_n and $\overline{\Delta}_{n-1}$ above are computable for every $n > 1$. Therefore, *average*(X) is incrementally updatable. Compare Δ_n and $\overline{\Delta}_{n-1}$ to equations 4.2 and 4.4 respectively. In Δ_n, *average*(X_{n-1}) is already known and *count*(X_n) can be incrementally computable from value of *count*(X_{n-1}). In $\overline{\Delta}_{n-1}$, *average*(X_n) is known and *count*(X_{n-1}) can be computed from value of *count*(X_n). ◁

4.3.4 Summative Aggregates

The vast majority of aggregates that are used or have a good potential of being used in decision making applications are those that perform some types of summation operations. We call such aggregates summative aggregates. In light of this observation, we focus our effort on making them incrementally updatable.

Let $X_n, Y_m, \cdots, Z_o$ be aggregate input sets whose respective current sizes are $n, m, \cdots, o$, and $x_i \in X_n$ $(1 \leq i \leq n)$, $y_j \in Y_m$ $(1 \leq j \leq m)$, $\cdots$, $z_k \in Z_o$ $(1 \leq k \leq o)$ be aggregate variables whose data types are numeric.

Summative Aggregate: Given aggregate input sets, $X_n, Y_m, \cdots, Z_o$, a *summative aggregate* is defined to be an algebraic aggregate that has summation operators ($\sum$s) in it as shown below:

$$
\mathcal{F}(X_n, Y_m, \cdots, Z_o) = \sum_{\rho=1}^{\mu} f(\psi_\rho, \rho, \mathcal{H}(X_n, Y_m, \cdots, Z_o)) \tag{4.5}
$$

where $f(\psi_\rho, \rho, \mathcal{H}(X_n, Y_m, \cdots, Z_o))$ is called *summation body*, ρ is called *index variable* and always starts from 1, μ is called *termination variable* and indicates the final index value, $\psi_\rho \in \{x_i, y_j, \cdots, z_k\}$, $(\rho, \mu) \in \{(i, n), (j, m), \cdots, (k, o)\}$, and $\mathcal{H}(X_n, Y_m, \cdots, Z_o)$ is a summative aggregate nested in $\mathcal{F}$ and differs from $\mathcal{F}$. If aggregate input sets

have the same size (i.e., $X_n, Y_n, \cdots, Z_n$), the definition of summative aggregate can be simpler as follows:

$$\mathcal{F}(X_n, Y_n, \cdots, Z_n) = \sum_{i=1}^{n} f(x_i, y_i, \cdots, z_i, i) \tag{4.6}$$

where the summation body $f(x_i, y_i, \cdots, z_i, i)$ may (recursively) contain other summative aggregates.

When the summation body of a summative aggregate contains one or more summation operators, the aggregate is called *nested summative aggregate.*

The summation body in a summative aggregate is defined to be an *aggregative polynomial*, defined below.

Aggregative polynomial. An *aggregative polynomial* is a polynomial that consists of zero or more plus' (+'s) as operators and *aggregative monomials* as operands.

Aggregative monomial. An *aggregative monomial* is a non-zero real constant multiplied by zero or more *factors.* A factor is either an aggregate variable, an index variable, a summative aggregate, or a parenthesized aggregative polynomial, any of which may be raised by a non-zero rational number or be an argument of transcendental functions on the real number such as the logarithm.[3] An aggregative monomial that does not contain another aggregate is called *atomic.* An aggregative monomial is sometimes called a *term.*

Example 4.3.4 Given aggregate variables, x, y, and z and index variables, i and j, the following are aggregative monomials:

$$1, \quad 1x_i, \quad -2.5\, i^2 x_i, \quad x_i^3\, y_i^{-1} \left(\textstyle\sum_{j=1}^{n} z_j\right)^2, \quad \sqrt{x_i - y_i}\ \ (= (x_i - y_i)^{1/2}), \quad \log_2 x_i.$$

[3] Ceiling ($\lceil\ \rceil$), floor ($\lfloor\ \rfloor$), and absolute ($|\ |$) functions are included too.

In $x_i^3\, y_i^{-1}\, (\sum_{j=1}^{n} z_j)^2$, factors are x_i^3, y_i^{-1}, and $(\sum_{j=1}^{n} z_j)^2$, whereas in $\log_2 x_i$, the sole factor is $\log_2 x_i$. Except for $x_i^3\, y_i^{-1}\, (\sum_{j=1}^{n} z_j)^2$, all the aggregative monomials are atomic. $\lhd$

Example 4.3.5 Given aggregate variables, x and y and index variables, i, j, and k, the following are aggregative polynomials:

$$x_i - (\sum_{j=1}^{n} x_j)/(\sum_{k=1}^{n} 1) \;\; (= x_i + (-(\sum_{j=1}^{n} x_j)(\sum_{k=1}^{n} 1)^{-1})), \quad x_i y_i, \quad 1.$$

Aggregative polynomial $x_i - (\sum_{j=1}^{n} x_j)/(\sum_{k=1}^{n} 1)$ has two terms (aggregative monomials), x_i and $-(\sum_{j=1}^{n} x_j)/(\sum_{k=1}^{n} 1)$, while the others have one term. $\lhd$

Example 4.3.6 All aggregates but *average*(X_n) shown in Example 4.3.1 are summative aggregates whose summative bodies are aggregative polynomials. $\lhd$

For a summative aggregate, one might be tempted to try to derive a positive delta and a negative delta directly as done in Example 4.3.3. However, deriving deltas for a nontrivial summative aggregate is not only arduous a work, but also such a derivation is not always possible. Furthermore, unless the derivation can be fully automated in an efficient way (which is very improbable), the aggregates cache cannot use the deltas to incrementally update cached aggregates. For summative aggregates, there exists a more efficient and simpler way of incremental update. We first consider non-nested summative aggregates in this subsection.

Lemma 3 *A summative aggregate whose summation body is an atomic aggregative monomial is incrementally updatable.*

Proof of Lemma 3 Obvious by the definition of summative aggregate.

$\square$

Lemma 4 *A summative aggregate whose summation body is an aggregative polynomial is incrementally updatable if all summative aggregates in aggregative monomials of the aggregative polynomial are incrementally updatable.*

Proof of Lemma 4 Assume any aggregative polynomial, $f_1(*) + f_2(*) + \cdots + f_v(*)$ where each $f_s(*)$, $1 \leq s \leq v$, is an aggregative monomial and $(*)$ represents any subset of aggregate variables and index variables defined in the original summative aggregate and need not be equal to each other. Then, by the distribution property of $\sum$ operator over additive (plus and minus) terms, the following equality holds:

$$\sum_{i=1}^{n}(f_1(*) + f_2(*) + \cdots + f_v(*)) = \sum_{i=1}^{n} f_1(*) + \sum_{i=1}^{n} f_2(*) + \cdots + \sum_{i=1}^{n} f_v(*).$$

Thus, if each $\sum_{i=1}^{n} f_s(*)$ can be updated incrementally, the original summative aggregate on the left-hand side of the equation above can be computed incrementally.

□

Now, the summative aggregate is extended to include more algebraic aggregates as follows.

Extended summative aggregate. Given a set of aggregate input sets, $\Omega = \{X_n, Y_m, \cdots, Z_o\}$ and a set of summative aggregates over subsets of Ω, $\mathcal{F}_1(\Omega_1)$, $\mathcal{F}_2(\Omega_2)$, $\cdots$, $\mathcal{F}_v(\Omega_v)$, $\Omega_s \subseteq \Omega$ $(1 \leq s \leq v)$, an aggregate $\mathcal{F}'$ over Ω is defined to be an *extended summative aggregate* if there exists an algebraic function g on the real number such that

$$\mathcal{F}'(X_n, Y_m, \cdots, Z_o) = g(\mathcal{F}_1(\Omega_1), \mathcal{F}_2(\Omega_2), \cdots, \mathcal{F}_v(\Omega_v)). \quad (4.7)$$

Note that in equation 4.7, the algebraic function g is not a summative aggregate by definition. g takes no argument of a set – its arguments are any number of scalar real values (since each $\mathcal{F}_s$ aggregate returns a scalar real number), whereas an aggregate is defined to take at least one *aggregate input set.*

Example 4.3.7 While the definition of extended summative aggregate covers a vast variety of algebraic aggregates that perform certain types of cumulative operations, perhaps the best known example of extended summative aggregate will be

$$average(X_n) = \sum_{i=1}^{n} x_i / \sum_{i=1}^{n} 1 = sum(X_n)/count(X_n).$$

Another example,

$$\log(\sum_{i=1}^{n} x_i) = \log(sum(X_n)).$$

$\triangleleft$

Example 4.3.8 This example shows an extended aggregate, $grand_average(X_n, Y_m)$ which is an average of two related aggregate input sets, X and Y.

$$\begin{aligned} grand_average(X_n, Y_m) &= \frac{grand_sum(X_n, Y_m)}{grand_count(X_n, Y_m)} \\ &= \frac{\sum_{i=1}^{n} x_i + \sum_{j=1}^{m} y_j}{\sum_{i=1}^{n} 1 + \sum_{j=1}^{m} 1}. \end{aligned}$$

$\triangleleft$

Lemma 5 *The extended summative aggregate of equation 4.7 is incrementally updatable if all $\mathcal{F}_s(\Omega_s)$'s $(1 \leq s \leq v)$ are incrementally updatable.*[4]

Proof of Lemma 5 Since g is a function, over a defined interval of its domain (see footnote 4), it should map a unique combination of input values (rather, every unique vector of input values) to the same one real number. Then, the mapping should remain the same no matter whether the input values are obtained through an incremental update or recomputed on the new aggregate input sets of the underlying summative aggregates. $\square$

[4] A function on the real number may have intervals of domain where the function is undefined (e.g., logarithm over negative values). Therefore, it is possible that for a certain g, g cannot be (incrementally) computed on some values of input summative aggregates. However, that is not because of the incremental update, but because of the definition of that specific aggregate.

Incidentally, it is interesting to note that if function g of equation 4.7 is a procedural function (i.e., non-algebraic function), Lemma 5 does not hold in general. A simple counter example: g being *max()*.

4.3.5 Binding of Variables

As shown in equation 4.5, a summative aggregate has two additional types of variables (other than aggregate variables), which are index variables and termination variables. Below, we describe how these variables are bound to a value or to other variable:

- The subscript of every aggregate variable is bound to an index variable; that is, an index variable is used to refer to a specific element in each aggregate input set.
- An instance of an index variable used in an aggregative monomial is bound to *the* index variable.
- The initial value of an index variable is always bound to 1 by definition.
- The final value of an index variable is bound to its matching termination variable.
- The termination variable of the outermost $\sum$ operator is either unbound (in most cases) or bound to a constant. If it is unbound, it should be bound to the current size (cardinality) of an associated aggregate input set when the aggregate is referenced by a query.
- The termination variable of an inner $\sum$ operator (i.e., in summation body) is either unbound, bound to an index variable of its an outer $\sum$ operator, or bound to a constant.

It should be noted that when a termination variable is bound to the cardinality of an aggregate input set, it is not necessary for the real value of the cardinality to be known, unless it is explicitly used in the summation body. In an incremental update, the purpose of termination variable is to know the latest element to be added to (or to be deleted from) an aggregate input set. On the other hand, if the cardinality is explicitly used in the summation body, it is interpreted as the simplest summative aggregate, *COUNT(*)* of the associated aggregate input set.

Scope of bindings. Each binding has a scope. A binding is in effect only within its scope. Scope of a summative aggregate is inside of its outermost $\sum$ operator. Scopes of the index and termination variables associated with a $\sum$ operator is inside of the summation operation, which is also called *the* $\sum$'s scope. Scope of an aggregate variable is the scope of its associated index variable. A nested summative aggregate has multiple nested scopes. For any two index variables, either one variable's scope properly contains the other variable's scope or the two scopes are independent. If two index variables' scopes are independent, the two index variables may be denoted by the same letter.

Example 4.3.9 The following are summative aggregates in which $\sum$ operators are nested (i.e., nested summative aggregates).

$$\mathcal{F}_1(X_n, Y_n, Z_n) = \sum_{i=1}^{n}(x_i \sum_{j=1}^{i}(y_j \sum_{k=1}^{j} i^2 z_k))$$

$$\mathcal{F}_2(X_n, Y_n, Z_n) = \sum_{i=1}^{n}(x_i \sum_{j=1}^{i}(y_j \sum_{k=1}^{i} i^2 z_k))$$

$$\mathcal{F}_3(X_n, Y_n, Z_n) = \sum_{i=1}^{n}(x_i(\sum_{j=1}^{n} y_j - \sum_{k=1}^{n} z_k)) = \sum_{i=1}^{n}(x_i(\sum_{j=1}^{n} y_j - \sum_{j=1}^{n} z_j))$$

In particular, in $\mathcal{F}_3$, scopes of $\sum_{j=1}^{n} y_j$ and $\sum_{k=1}^{n} z_k$ are independent. So, both summative aggregates can be represented using the same index variable j. $\lhd$

4.3.6 Decomposition of Summative Aggregates

In the aggregate cache, the basic unit cached is a summation over an aggregative monomial. It is, therefore, more beneficial to decompose a complex summative aggregate into a set of smaller component summative aggregates and store their values in the aggregate cache. When value of the original summative aggregate is necessary, it can be easily restored from the values of the cached component aggregates. This approach facilitates sharing cached component aggregates among many summative aggregates. Furthermore, the decomposition plays a more important role in the aggregate cache. While decomposing the original summative aggregate and normalizing the decomposed summative aggregates, it becomes known whether the original summative aggregate can be incrementally updated or not.[5] If the original summative aggregate is not incrementally updatable, it can still be cached but the cached value should be invalidated if changes in base databases affect it.

In principle, the decomposition process is done in two steps: expansion of the summation body of a given summative aggregate and distribution of $\sum$ operators over the expanded summation body. If a given summative aggregate is nested (i.e., it contains other summative aggregates), the summation body is recursively expanded until no further expansion is possible. Then, $\sum$ operators are distributed over the whole expansion. By the distribution property of $\sum$ operator over additive terms, sum of the decomposed summative aggregates is equivalent to the original summative aggregate.

In order to describe the expansion procedure precisely, let's represent an arbitrary aggregative monomial $h(*)$ as follows:

$$h(*) = ca_1^{p_1} a_2^{p_2} \cdots a_t^{p_t} \tag{4.8}$$

[5] The normalization will be discussed in Section 4.3.7

where $*$ is a union of aggregate input sets and index variables, c is a non-zero real constant, each $a_l^{p_l}$ $(1 \leq l \leq t)$ is a factor, assuming that if $t = 0$, no factor is present in the aggregative monomial,[6] and each p_l $(1 \leq l \leq t)$ is a non-zero rational number. If $a_l^{p_l}$ denotes a factor other than a transcendental function, it means a_l raised by p_l. Otherwise, $a_l^{p_l}$ simply denotes a transcendental function. (See Section 4.3.4 and Example 4.3.4.)

In order to expand the aggregative monomial $h(*)$ of equation 4.8, each factor $a_l^{p_l}$ $(1 \leq l \leq t)$ is expanded first. If it is expanded into multiple terms, $h(*)$ is expanded accordingly into multiple aggregative monomials by usual algebraic manipulations.

Example 4.3.10 If $a_1^{p_1}$ and $a_t^{p_t}$ in equation 4.8 are expanded into $(a_{1_1}^{p_{1_1}} + a_{1_2}^{p_{1_2}})$ and $(a_{t_1}^{p_{t_1}} - a_{t_2}^{p_{t_2}})$ respectively, $h(*)$ is expanded into four aggregative monomials as follows:

$$
\begin{aligned}
h(*) &= c a_1^{p_1} a_2^{p_2} \cdots a_t^{p_t} \\
&= c a_{1_1}^{p_{1_1}} a_2^{p_2} \cdots a_t^{p_t} + c a_{1_2}^{p_{1_2}} a_2^{p_2} \cdots a_t^{p_t} \\
&= c a_{1_1}^{p_{1_1}} a_2^{p_2} \cdots a_{t_1}^{p_{t_1}} - c a_{1_1}^{p_{1_1}} a_2^{p_2} \cdots a_{t_2}^{p_{t_2}} \\
&\quad + c a_{1_2}^{p_{1_2}} a_2^{p_2} \cdots a_{t_1}^{p_{t_1}} - c a_{1_2}^{p_{1_2}} a_2^{p_2} \cdots a_{t_2}^{p_{t_2}}.
\end{aligned}
$$

$\lhd$

Expansion of factor. Expansion of a factor $a_l^{p_l}$ in equation 4.8 is obtained by a factor expansion procedure below:

1. If $a_l^{p_l}$ denotes a transcendental function, expansion of $a_l^{p_l}$ is $a_l^{p_l}$ itself. That is, a transcendental function is not (or can't be) expanded.[7]

[6] Hereafter, for a sequence of any entities, $\alpha_1, \alpha_2, \cdots, \alpha_t$ $(t \geq 0)$, if $t = 0$, the sequence is assumed to be null (i.e., no entity in the sequence).

[7] For some transcendental functions, there are a few cases in which expansion is possible. For example, $\log(x_i/y_i)$ can be expanded into $\log x_i - \log y_i$. However, in most cases such an expansion is not possible. Therefore, in our work we do not expand transcendental functions.

2. If p_l (of $a_l^{p_l}$) is *not* a positive integer, expansion of factor $a_l^{p_l}$ is $a_l^{p_l}$ itself.[8]

3. Otherwise, a_l (of $a_l^{p_l}$) is expanded first as follows:

   (a) If a_l is a parenthesized aggregative polynomial, its component aggregative monomials are recursively expanded first and their results are added, yielding $h_{l_1}(*') + h_{l_2}(*') + \cdots + h_{l_w}(*')$ $(w \geq 1)$, where $*'$ is a subset of $*$ and each $*'$ need not be equal to each other. Then, expansion of a_l is $h_{l_1}(*') + h_{l_2}(*') + \cdots + h_{l_w}(*')$.

   (b) If a_l is a summative aggregate $\sum_{\rho=1}^{\mu} f_l(*')$, its parenthesized summation body $(f_l(*'))$ is recursively expanded first, yielding $h_{l_1}(*') + h_{l_2}(*') + \cdots + h_{l_w}(*')$ $(w \geq 1)$. Then, expansion of a_l is $\sum_{\rho=1}^{\mu} h_{l_1}(*') + \sum_{\rho=1}^{\mu} h_{l_2}(*') + \cdots + \sum_{\rho=1}^{\mu} h_{l_w}(*')$.

   (c) If a_l is an index variable or an aggregate variable, expansion of a_l is a_l itself.

   After expanding a_l, if the expansion of a_l is a single term, expansion of factor $a_l^{p_l}$ is $a_l^{p_l}$. Otherwise, expansion of factor $a_l^{p_l}$ is obtained by applying the multinomial theorem (see below) to the expansion of a_l. $\lhd$

After expansion of a summation body is completed, if there exist factors in the form of $(c_s a_{s_1}^{p_{s_1}} a_{s_2}^{p_{s_2}} \cdots a_{s_r}^{p_{s_r}})^{p_q}$ $(r \geq 1)$ in the expansion, they are converted into $c_s^{p_q} a_{s_1}^{p_{s_1} p_q} a_{s_2}^{p_{s_2} p_q} \cdots a_{s_r}^{p_{s_r} p_q}$.

Decomposition of summative aggregates. Suppose a summative aggregate (not extended), $\sum_{i=1}^{n} f(*)$, where $*$ is a union of aggregate input sets and index variable i. In order to apply the expansion procedure described so far, the summation body

[8]Again, for an expression $(a_1 + a_2 + \cdots + a_n)^p$, if the exponent p is not a positive (rather, non-negative) integer, it is generally not possible to expand the expression in finite steps.

$f(*)$ (which is an aggregative polynomial) is changed into an aggregative monomial by parenthesizing it as follows:

$$\sum_{i=1}^{n} f(*) = \sum_{i=1}^{n} (f(*)). \tag{4.9}$$

Then, after expanding the new summation body, let the result be:

$$(f(*)) = h_1(*') + h_2(*') + \cdots + h_v(*') \quad (v \geq 1) \tag{4.10}$$

where each $h_s(*')$ $(1 \leq s \leq v)$ is an aggregative monomial, $*'$ is a subset of $*$, and each $*'$ need not be equal to each other. Then, by distributing $\sum$ operator over the right-hand side of equation 4.10, the following decomposed summative aggregate is obtained:

$$\sum_{i=1}^{n} f(*) = \sum_{i=1}^{n} h_1(*') + \sum_{i=1}^{n} h_2(*') + \cdots + \sum_{i=1}^{n} h_v(*'). \tag{4.11}$$

<u>Decomposition of extended summative aggregates.</u> For an extended summative aggregate, $\mathcal{F}(*) = g(\mathcal{F}_1(*'), \mathcal{F}_2(*'), \cdots, \mathcal{F}_v(*'))$ $(v \geq 1)$, each argument summative aggregate $\mathcal{F}_s(*')$ $(1 \leq s \leq v)$ is decomposed first, yielding $\mathcal{F}'_s(*')$. Then, the original argument summative aggregates are substituted with the decomposed ones, yielding $\mathcal{F}(*) = g(\mathcal{F}'_1(*'), \mathcal{F}'_2(*'), \cdots, \mathcal{F}'_v(*'))$.

Any monomials (not only aggregative monomials) in the form of $(a_1+a_2+\cdots+a_n)^p$ where p is a positive integer can be expanded using the *multinomial theorem* which is an extension of the *binomial theorem.*[9] The binomial theorem and multinomial theorem are presented below.

<u>Binomial theorem.</u> Let p be a positive integer. Then for all x and y,

$$(x+y)^p = \sum_{k=0}^{p} \binom{p}{k} x^k y^{p-k}$$

[9]In fact, the multinomial theorem (binomial theorem as well) holds even when the exponent p is a non-zero rational number. However, in general, expanding such a monomial results in an infinite series, which is of no use for the aggregate cache.

where $\begin{pmatrix} p \\ k \end{pmatrix} = p! \,/\, k!\,(p-k)!$.

<u>Multinomial theorem.</u> Let p be a positive integer. Then for all $x_1, x_2, \cdots, x_t$,

$$(x_1 + x_2 + \cdots + x_t)^p = \sum \begin{pmatrix} p \\ p_1 p_2 \cdots p_t \end{pmatrix} x_1^{p_1} x_2^{p_2} \cdots x_t^{p_t},$$

where the summation extends over all possible sequence of non-negative integers $p_1, p_2, \cdots, p_t$ with $p_1 + p_2 + \cdots + p_t = p$, and $\begin{pmatrix} p \\ p_1 p_2 \cdots p_t \end{pmatrix} = p! \,/\, p_1!\, p_2! \cdots p_t!$.

<u>Example 4.3.11</u> When $(x_1 + x_2 + x_3 + x_4 + x_5)^6$ is expanded, the coefficient of $x_1^3 x_2 x_4^2 x_5$ $(= x_1^3 x_2^1 x_3^0 x_4^2 x_5^1)$ equals

$$\begin{pmatrix} 6 \\ 3\ 1\ 0\ 2\ 1 \end{pmatrix} = \frac{6!}{3!\,1!\,0!\,2!\,1!} = 60.$$

When $(4x_1 - 2x_2 + 3x_3)^5$ is expanded, the coefficient of $x_1^2 x_2^3 x_3$ equals

$$\begin{pmatrix} 5 \\ 2\ 3\ 1 \end{pmatrix} 4^2 (-2)^3 3^1 = -3840.$$

$\lhd$

Using the multinomial theorem, it is possible to expand any aggregative monomial raised to a *positive integer power*; if its component monomials are raised, they can be recursively expanded. From a practical viewpoint, however, it is not a good idea to expand an aggregative monomial which is raised by more than 3, since the number of total monomials produced increases exponentially. In this regard, we expand (recursively) only those aggregative monomials raised by 3 or less.

<u>Example 4.3.12</u> Consider the following summative aggregate:

$$\sum_{i=1}^{n} (x_i - y_i)^2$$

where the summation body is $(x_i - y_i)^2$. Assume that the current aggregate input sets are X_{n-1} and Y_{n-1} and aggregate value on these input sets, $\sum_{i=1}^{n-1} (x_i - y_i)^2$ is

stored in the aggregate cache. Then, upon insertions of x_n and y_n, the new aggregate value can be computed by adding value of $(x_n - y_n)^2$ to the stored aggregate value. The summative aggregate is decomposed as follows:

$$\sum_{i=1}^{n}(x_i - y_i)^2 = \sum_{i=1}^{n}(x_i^2 - 2x_iy_i + y_i^2) = \sum_{i=1}^{n} x_i^2 - 2\sum_{i=1}^{n} x_iy_i + \sum_{i=1}^{n} y_i^2.$$

Instead of storing $\sum_{i=1}^{n-1}(x_i - y_i)^2$ directly, it is much better to store $\sum_{i=1}^{n-1} x_i^2$, $\sum_{i=1}^{n-1} x_iy_i$, and $\sum_{i=1}^{n-1} y_i^2$ so that other summative aggregates too can make use of the store values.

Similarly, a more complex summative aggregate is decomposed as follows:

$$\begin{aligned}
&\sum_{i=1}^{n}(2x_i - y_i + z_i)^3 \\
&\quad = \sum_{i=1}^{n}(8x_i^3 - y_i^3 + z_i^3 - 12x_i^2y_i + 12x_i^2z_i + 6x_iy_i^2 \\
&\qquad + 3y_i^2z_i + 6x_iz_i^2 - 3y_iz_i^2 - 12x_iy_iz_i) \\
&\quad = 8\sum_{i=1}^{n} x_i^3 - \sum_{i=1}^{n} y_i^3 + \sum_{i=1}^{n} z_i^3 - 12\sum_{i=1}^{n} x_i^2y_i + 12\sum_{i=1}^{n} x_i^2z_i + 6\sum_{i=1}^{n} x_iy_i^2 \\
&\qquad + 3\sum_{i=1}^{n} y_i^2z_i + 6\sum_{i=1}^{n} x_iz_i^2 - 3\sum_{i=1}^{n} y_iz_i^2 - 12\sum_{i=1}^{n} x_iy_iz_i.
\end{aligned}$$

This summative aggregate shows the rapid increase of the number of aggregative monomials as the exponent increases. $\lhd$

4.3.7 Normalization of Summative Aggregates

Normalization of a summative aggregate is a process of simplifying the summative aggregate. The simplest normalization is to pull a constant, multiplying the summation body, out of the scope of its summation operator. For instance, $\sum_{i=1}^{n} -4x_i = -4\sum_{i=1}^{n} x_i$. By normalizing a summative aggregate in this way, $\sum_{i=1}^{n} x_i$ can be cached instead of $\sum_{i=1}^{n} -4x_i$ so that cache operations can be more efficient. In fact, the normalization plays a more profound role than making cache operations efficient. It enables certain types of summative aggregates to be incrementally updated, which otherwise cannot be. In this subsection we extend this simple idea in order to normalize complex nested summative aggregates.

As summative aggregates are nested, some unbound entities (variables and component summative aggregates) can be present within the scope of a summative aggregate. These unbound entities hinder the process of incremental update of summative aggregates since their values are not known at the time of an incremental update. There are several entities that can be unbound within a component summative aggregate of a nested summative aggregate.

1. An index variable of an outer summation operator.
2. An aggregate variable indexed by an index variable of an outer summation operator.
3. A summative aggregate whose termination variable is bound to the cardinality of an aggregate input set.
4. A summative aggregate whose termination variable is an index variable of an outer summation operator.

The goal of normalization is to make, recursively, every summative aggregate contain only those entities bound to its index variable by moving unbound entities plus a constant multiplying the aggregative monomial out of the scope of the summative aggregate. Therefore, once a normalization is completed, no unbound entities shall remain within any summative aggregate's scope.

Normalization Process

Without loss of generality, it is assumed that a summative aggregate is decomposed maximally (i.e., as far as possible) before normalized. After the decomposition, each decomposed summative aggregate should have a summation body of an aggregative monomial. Then, the normalization is performed against each decomposed summative aggregate. The following is a decomposed summative aggregate to

be normalized:

$$\sum_{\rho=1}^{\mu} c a_1^{p_1} a_2^{p_2} \cdots a_t^{p_t} \quad (t \geq 0) \tag{4.12}$$

where ρ is an index variable, μ is a termination variable, c is a non-zero real constant, each $a_l^{p_l}$ $(1 \leq l \leq t)$ is a *factor*, and each p_l $(1 \leq l \leq t)$ is a non-zero rational number. (See equation 4.8.)

Given an index variable ρ, a factor is *fully bound* to ρ if ρ is the sole index variable that appears in the factor. A factor is *partially bound* to ρ if ρ and other index variables appear in the factor. If ρ does not appear in a factor, the factor is *unbound* to ρ.

Normalization of a summative aggregate 4.12 takes two passes:

1. For each factor $a_l^{p_l}$, if a_l (of $a_l^{p_l}$) is a summative aggregate, it is normalized first, yielding $c_l^{p_l} a_{l_1}^{p_l} a_{l_2}^{p_l} a_{l_{(t_l)}}^{p_l}$. After all component summative aggregates are normalized, let the result be the following:

$$\sum_{\rho=1}^{\mu} (c c_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{1_1}^{p_1} a_{1_2}^{p_1} \cdots a_{1_{(t_1)}}^{p_1} a_{2_1}^{p_2} a_{2_2}^{p_2} \cdots a_{2_{(t_2)}}^{p_2} \cdots a_{t_1}^{p_t} a_{t_2}^{p_t} \cdots a_{t_{(t_t)}}^{p_t} \tag{4.13}$$

   where each $t_s \geq 1$ $(1 \leq s \leq t)$, and let $q = t_1 + t_2 + \cdots + t_t$.

2. Unbound factors in the summative aggregate 4.13 are moved out of the scope of the $\sum$ operator as follows:

$$(c c_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{u_1}^{p_{u_1}} a_{u_2}^{p_{u_2}} \cdots a_{u_v}^{p_{u_v}} \sum_{\rho=1}^{\mu} a_{b_1}^{p_{b_1}} a_{b_2}^{p_{b_2}} \cdots a_{b_w}^{p_{b_w}} \quad (v \geq 0,\ w \geq 0) \tag{4.14}$$

   where each $a_u^{p_u}$ is an unbound factor to ρ, each $a_b^{p_b}$ is a either fully or partially bound factor to ρ, and $v + w = q$.[10]

In the transformed summative aggregate 4.14, if the right-hand side of $\sum$ operator contains *only* bound factor to ρ and all component summative aggregates, if any, are

[10] Note that by the normalization a summative aggregate can be transformed to an extended summative aggregate. (See $\mathcal{F}_2$ and $\mathcal{F}_3$ in Example 4.3.13.)

normalized, the summative aggregate is called *normalized.* Otherwise, the original summative aggregate 4.12 is called *unnormalizable.*

Theorem 3 *A normalized summative aggregate is equivalent to the original summative aggregate, that is:*

$$\sum_{\rho=1}^{\mu} ca_1^{p_1} a_2^{p_2} \cdots a_t^{p_t} = (cc_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{u_1}^{p_{u_1}} a_{u_2}^{p_{u_2}} \cdots a_{u_q}^{p_{u_q}} \sum_{\rho=1}^{\mu} a_{b_1}^{p_{b_1}} a_{b_2}^{p_{b_2}} \cdots a_{b_r}^{p_{b_r}}.$$

Proof of Theorem 3 We first show that summative aggregates 4.13 and 4.14 are equivalent, that is:

$$\sum_{\rho=1}^{\mu} (cc_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{1_1}^{p_1} a_{1_2}^{p_1} \cdots a_{1_{(t_1)}}^{p_1} a_{2_1}^{p_2} a_{2_2}^{p_2} \cdots a_{2_{(t_2)}}^{p_2} \cdots a_{t_1}^{p_t} a_{t_2}^{p_t} \cdots a_{t_{(t_t)}}^{p_t}$$
$$= (cc_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{u_1}^{p_{u_1}} a_{u_2}^{p_{u_2}} \cdots a_{u_v}^{p_{u_v}} \sum_{\rho=1}^{\mu} a_{b_1}^{p_{b_1}} a_{b_2}^{p_{b_2}} \cdots a_{b_w}^{p_{b_w}}. \tag{4.15}$$

To show the equivalence above, we put $(cc_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{u_1}^{p_{u_1}} a_{u_2}^{p_{u_2}} \cdots a_{u_v}^{p_{u_v}}$ of summative aggregate 4.14 back inside $\sum$ operator. Then, the following equality should hold since the left-hand side and the right-hand side of the equation are literally equivalent $(q = v + w)$:

$$\sum_{\rho=1}^{\mu} (cc_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{1_1}^{p_1} a_{1_2}^{p_1} \cdots a_{1_{(t_1)}}^{p_1} a_{2_1}^{p_2} a_{2_2}^{p_2} \cdots a_{2_{(t_2)}}^{p_2} \cdots a_{t_1}^{p_t} a_{t_2}^{p_t} \cdots a_{t_{(t_t)}}^{p_t}$$
$$= \sum_{\rho=1}^{\mu} (cc_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{u_1}^{p_{u_1}} a_{u_2}^{p_{u_2}} \cdots a_{u_v}^{p_{u_v}} a_{b_1}^{p_{b_1}} a_{b_2}^{p_{b_2}} \cdots a_{b_w}^{p_{b_w}}.$$

Letting $(cc_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{u_1}^{p_{u_1}} a_{u_2}^{p_{u_2}} \cdots a_{u_v}^{p_{u_v}}$ be ϕ and expanding $\sum$ operator of the right-hand side of the equation above, the following equation is obtained:

$$\sum_{\rho=1}^{\mu} (cc_1^{p_1} c_2^{p_2} \cdots c_t^{p_t}) a_{u_1}^{p_{u_1}} a_{u_2}^{p_{u_2}} \cdots a_{u_v}^{p_{u_v}} a_{b_1}^{p_{b_1}} a_{b_2}^{p_{b_2}} \cdots a_{b_w}^{p_{b_w}}$$
$$= \phi_1 (a_{b_1}^{p_{b_1}})_1 (a_{b_2}^{p_{b_2}})_1 \cdots (a_{b_w}^{p_{b_w}})_1 + \phi_2 (a_{b_1}^{p_{b_1}})_2 (a_{b_2}^{p_{b_2}})_2 \cdots (a_{b_w}^{p_{b_w}})_2$$
$$+ \cdots + \phi_\mu (a_{b_1}^{p_{b_1}})_\mu (a_{b_2}^{p_{b_2}})_\mu \cdots (a_{b_w}^{p_{b_w}})_\mu$$

where ϕ_ρ or $(a_b^{p_b})_\rho$ $(1 \le \rho \le \mu)$ represents an expression in which all occurrences of index variable ρ are substituted by a value of ρ. However, since ϕ is unbound to ρ

(i.e., there is no occurrence of ρ in ϕ), all ϕ_ρ's ($1 \leq \rho \leq \mu$) should be the same in the above equation. Therefore, we get the following equation, thereby proving the equality of the equation 4.15:

$$\begin{aligned}\phi\,[\,(a_{b_1}^{p_{b_1}})_1(a_{b_2}^{p_{b_2}})_1\cdots(a_{b_w}^{p_{b_w}})_1+(a_{b_1}^{p_{b_1}})_2(a_{b_2}^{p_{b_2}})_2\cdots(a_{b_w}^{p_{b_w}})_2 \\ +\cdots+(a_{b_1}^{p_{b_1}})_\mu(a_{b_2}^{p_{b_2}})_\mu\cdots(a_{b_w}^{p_{b_w}})_\mu\,] \\ =(cc_1^{p_1}c_2^{p_2}\cdots c_t^{p_t})a_{u_1}^{p_{u_1}}a_{u_2}^{p_{u_2}}\cdots a_{u_q}^{p_{u_q}}\sum_{\rho=1}^{\mu}a_{b_1}^{p_{b_1}}a_{b_2}^{p_{b_2}}\cdots a_{b_r}^{p_{b_r}}.\end{aligned}$$

The next step is to show the equivalence between summative aggregates 4.12 and 4.13. If any factor $a_l^{p_l}$ in summative aggregate 4.12 is a power of a summative aggregate (i.e., a_l is a summative aggregate), the factor is normalized to $c_l^{p_l}a_{l_1}^{p_l}a_{l_2}^{p_l}a_{l_{(t_l)}}^{p_l}$. Then, the proof shown above implies the equivalence of $a_l^{p_l}$ and $c_l^{p_l}a_{l_1}^{p_l}a_{l_2}^{p_l}a_{l_{(t_l)}}^{p_l}$. (The equation 4.15 directly proves $a_l = c_l a_{l_1} a_{l_2} \cdots a_{l_{(t_l)}}$.) If the summation body of a_l contains other summative aggregates again, they are (recursively) equivalently normalized before a_l is normalized. Therefore, we can conclude that summative aggregates 4.12 and 4.13 are equivalent, thereby proving Theorem 3.

□

Normalization of extended summative aggregates. In order to normalize an extended summative aggregate, $\mathcal{F}(*) = g(\mathcal{F}_1(*'), \mathcal{F}_2(*'), \cdots, \mathcal{F}_v(*'))$ ($v \geq 1$), it is decomposed first, yielding $\mathcal{F}(*) = g(\mathcal{F}'_1(*'), \mathcal{F}'_2(*'), \cdots, \mathcal{F}'_v(*'))$. Then, each decomposed argument summative aggregate $\mathcal{F}'_s(*')$ ($1 \leq s \leq v$) is normalized, yielding $\mathcal{F}''_s(*')$. Then, the original argument summative aggregates are substituted with the normalized ones, yielding $\mathcal{F}(*) = g(\mathcal{F}''_1(*'), \mathcal{F}''_2(*'), \cdots, \mathcal{F}''_v(*'))$.

Example 4.3.13 The following nested summative aggregates are converted to their respective normalized forms.

$$\mathcal{F}_1(X_n, Y_n) = \sum_{i=1}^{n}(x_i \sum_{j=1}^{i} 2i^3 y_j) = 2\sum_{i=1}^{n}(i^3 x_i \sum_{j=1}^{i} y_j)$$

In $\mathcal{F}_1$, 2 and i^3 are unbound in $\sum_{j=1}^{i} 2i^3 y_j$. So, these entities are moved out of index variable j's scope. As a constant, 2 is further moved out of index variable i's scope.

$$\mathcal{F}_2(X_n, Y_m) = \sum_{i=1}^{n}(x_i \sum_{j=1}^{m} y_j) = (\sum_{j=1}^{m} y_j)(\sum_{i=1}^{n} x_i)$$

In $\mathcal{F}_2$, the component aggregate $\sum_{j=1}^{m} y_j$ is an unbound summative aggregate. Thus, it can be treated as a constant from the viewpoint of any surrounding summative aggregates. So is the result. Note also that the normalized $\mathcal{F}_2$ is no longer an ordinary summative aggregate, but an extended summative aggregate. (The same for $\mathcal{F}_3$ below.)

$$\begin{aligned}
\mathcal{F}_3(X_n, Y_n, Z_n) &= \sum_{i=1}^{n}(x_i \sum_{j=1}^{i}(y_j \sum_{k=1}^{n} j z_k)) \\
&= \sum_{i=1}^{n}(x_i(\sum_{k=1}^{n} z_k) \sum_{j=1}^{i} j y_j) \\
&= (\sum_{k=1}^{n} z_k)(\sum_{i=1}^{n}(x_i \sum_{j=1}^{i} j y_j))
\end{aligned}$$

In $\mathcal{F}_3$, the component summative aggregate $\sum_{k=1}^{n} j z_k$ is unbound and in itself, it contains an unbound index variable j. So, it moves j up to j's scope and it itself moves out of the outermost i's scope.

$$\mathcal{F}_4(X_n, Y_n) = \sum_{i=1}^{n}(x_i - \sum_{j=1}^{n} y_j)^{1/2}$$

In $\mathcal{F}_4$, on the other hand, even though $\sum_{j=1}^{n} y_j$ is unbound, there is no known way of moving the unbound summative aggregate out of the outer index variable i's scope. Therefore, $\mathcal{F}_4$ is unnormalizable. $\triangleleft$

Example 4.3.14 In this example, as an extended summative aggregate, the standard deviation is normalized. The standard deviation has two argument summative aggregates, $\sum_{i=1}^{n}(x_i - \overline{x})^2$ and n (i.e., $\sum_{i=1}^{n} 1$), which are decomposed in equations 4.18 and 4.19 below. Then, the decomposed argument summative aggregates are normalized in equation 4.20. Note that in the normalized aggregate, there are only three unique summative aggregates, $\sum_{i=1}^{n} x_i^2$, $\sum_{i=1}^{n} x_i$, and $\sum_{i=1}^{n} 1$.

$$\sigma = \mathcal{F}(X_n) = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}} \tag{4.16}$$

$$= \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \frac{\sum_{j=1}^{n} x_j}{\sum_{j=1}^{n} 1}\right)^2}{(\sum_{i=1}^{n} 1) - 1}} \tag{4.17}$$

$$= \sqrt{\frac{\sum_{i=1}^{n}\left(x_i^2 - 2x_i\frac{\sum_{j=1}^{n} x_j}{\sum_{j=1}^{n} 1} + \left(\frac{\sum_{j=1}^{n} x_j}{\sum_{j=1}^{n} 1}\right)^2\right)}{(\sum_{i=1}^{n} 1) - 1}} \tag{4.18}$$

$$= \sqrt{\frac{\sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n}\left(-2x_i\frac{\sum_{j=1}^{n} x_j}{\sum_{j=1}^{n} 1}\right) + \sum_{i=1}^{n}\left(\frac{\sum_{j=1}^{n} x_j}{\sum_{j=1}^{n} 1}\right)^2}{(\sum_{i=1}^{n} 1) - 1}} \tag{4.19}$$

$$= \sqrt{\frac{\sum_{i=1}^{n} x_i^2 - 2\,\frac{\sum_{j=1}^{n} x_j}{\sum_{j=1}^{n} 1}\sum_{i=1}^{n} x_i + \frac{\left(\sum_{j=1}^{n} x_j\right)^2}{\left(\sum_{j=1}^{n} 1\right)^2}\sum_{i=1}^{n} 1}{(\sum_{i=1}^{n} 1) - 1}} \tag{4.20}$$

$\triangleleft$

4.3.8 Incremental-Updatability of Nested Summative Aggregates

As Lemma 3 states, a summative aggregate is incrementally updatable if its summation body is an atomic aggregative monomial. In case its summation body is an aggregative polynomial, by Lemma 4, the summative aggregative is incrementally updatable, if each aggregative monomial in the aggregative polynomial is atomic. For *nested summative aggregates*, however, it turns out that not all of them are incrementally updatable. In this subsection we identify those nested aggregates that are not incrementally updatable. Also, for those that are incrementally updatable, we present how to compute them.

Summative aggregates shown in Example 4.3.13 are quite suggestive of which nested summative aggregates are incrementally updatable and which are not. In short, all $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$ in the example are incrementally updatable, whereas $\mathcal{F}_4$ is not.

Let's compare $\mathcal{F}_2$ and $\mathcal{F}_4$ first. It should be noted that without the normalization, even $\mathcal{F}_2$ cannot be incrementally updated. $\mathcal{F}_2$ has a summation body of $(x_i \sum_{j=1}^{m} y_j)$ that nests an unbound summative aggregate $\sum_{j=1}^{m} y_j$. One cannot save the value of $\mathcal{F}_2(X_{n-1}, Y_{m-1}) = \sum_{i=1}^{n-1}(x_i \sum_{j=1}^{m-1} y_j)$ and use that value to compute $\mathcal{F}_2(X_n, Y_m)$ on insertion of (x_n, y_m), due to the following inequality:

$$
\begin{aligned}
\mathcal{F}_2(X_n, Y_m) &= \sum_{i=1}^{n}(x_i \sum_{j=1}^{m} y_j) \\
&\neq \sum_{i=1}^{n-1}(x_i \sum_{j=1}^{m-1} y_j) + (x_n \sum_{j=1}^{m} y_j) = \mathcal{F}_2(X_{n-1}, Y_{m-1}) + (x_n \sum_{j=1}^{m} y_j).
\end{aligned}
$$

Only after normalizing $\mathcal{F}_2$, the following equality is obtained, thereby making incremental update possible:

$$
\begin{aligned}
\mathcal{F}_2(X_n, Y_m) &= (\sum_{i=1}^{n} x_i)(\sum_{j=1}^{m} y_j) \\
&= (\sum_{i=1}^{n-1} x_i)(\sum_{j=1}^{m-1} y_j) + (x_n)(y_m) = \mathcal{F}_2(X_{n-1}, Y_{m-1}) + (x_n)(y_m).
\end{aligned}
$$

The reason why $\mathcal{F}_2$ is not incrementally updatable before the normalization is that the unbound summative aggregate $\sum_{j=1}^{m} y_j$ within the scope of index variable i should act as a *constant* while it takes part in the outer summation operator's operation, but without the normalization, its value varies in successive incremental updates. By normalizing $\mathcal{F}_2$, however, the original aggregate is transformed to a multiplication of two summative aggregates, each of which can be independently incrementally updated. Then, by Lemma 5, the normalized summative aggregate is incrementally updatable.

On the other hand, as mentioned in Example 4.3.13, $\mathcal{F}_4$ cannot be normalized. Similar to $\mathcal{F}_2$ before the normalization, $\mathcal{F}_4$ has the following inequality:

$$
\begin{aligned}
\mathcal{F}_4(X_n, Y_n) &= \sum_{i=1}^{n}(x_i - \sum_{j=1}^{n} y_j)^{1/2} \\
&\neq \sum_{i=1}^{n-1}(x_i - \sum_{j=1}^{n-1} y_j)^{1/2} + (x_n - \sum_{j=1}^{n} y_j)^{1/2}
\end{aligned}
$$

$$= \mathcal{F}_4(X_{n-1}, Y_{n-1}) + (x_n - \sum_{j=1}^{n} y_j)^{1/2}.$$

Since $\mathcal{F}_4$ cannot be normalized, there is no chance of $\mathcal{F}_4$ being incrementally updatable.

$\mathcal{F}_1$ and $\mathcal{F}_3$ are incrementally updatable as mentioned. Unlike $\mathcal{F}_2$, however, these summative aggregates are still nested even after the normalization. Nonetheless, $\mathcal{F}_1$ has the following equality:

$$\begin{aligned}
\mathcal{F}_1(X_n, Y_n) &= 2\sum_{i=1}^{n}(i^3 x_i \sum_{j=1}^{i} y_j) \\
&= (2\sum_{i=1}^{n-1}(i^3 x_i \sum_{j=1}^{i} y_j)) + (2n^3 x_n \sum_{j=1}^{n} y_j) \\
&= \mathcal{F}_1(X_{n-1}, Y_{n-1}) + (2n^3 x_n \sum_{j=1}^{n} y_j)
\end{aligned}$$

The above equation itself can be served as a proof by induction of the incremental updatability. When $i = 1$ (base), the equality holds trivially, letting $\mathcal{F}_1(X_0, Y_0) = 0$. Assuming $\mathcal{F}_1(X_{n-1}, Y_{n-1})$ is incrementally updatable (induction hypothesis), $\mathcal{F}_1(X_n, Y_n)$ is incrementally updatable since $\sum_{j=1}^{n} y_j$ too is incrementally updatable. Thus, proved.

Now consider a little experiment with $\mathcal{F}_4$. $\mathcal{F}_4$ is modified to $\mathcal{F}_5$ as below so that the inner summative aggregate becomes bound:

$$\begin{aligned}
\mathcal{F}_5(X_n, Y_n) &= \sum_{i=1}^{n}(x_i - \sum_{j=1}^{i} y_j)^{1/2} \\
&= \sum_{i=1}^{n-1}(x_i - \sum_{j=1}^{i} y_j)^{1/2} + (x_n - \sum_{j=1}^{n} y_j)^{1/2} \\
&= \mathcal{F}_5(X_{n-1}, Y_{n-1}) + (x_n - \sum_{j=1}^{n} y_j)^{1/2}.
\end{aligned}$$

A similar induction can be used to prove correctness of the above equality. Therefore, $\mathcal{F}_5$ is incrementally updatable.

For every summative aggregate, it is true that if the summative aggregate contains any unbound variables or unbound summative aggregates, the summative aggregate in question cannot be incrementally updated.

Lemma 6 *A summative aggregate that contains only bound entities is incrementally updatable, if, recursively, all its component summative aggregates, if any, contain only bound entities.*

Proof of Lemma 6 For any summative aggregate,

$$\mathcal{F}(X_n, Y_n, \cdots, Z_n) = \sum_{i=1}^{n} f(x_i, y_i, \cdots, z_i, i).$$

Proof by induction.

(Below we show only the case of insertion. For the case of deletion, similar induction steps can be easily applied.)

1. When $i = 1$ (base):

   Letting $\mathcal{F}(X_0, Y_0, \cdots, Z_0) = 0$, incremental update on the first insertion $(x_1, y_1, \cdots, z_1)$,

   $$\mathcal{F}(X_1, Y_1, \cdots, Z_1) = \mathcal{F}(X_0, Y_0, \cdots, Z_0) + f(x_1, y_1, \cdots, z_1, 1).$$

   Thus, the summative aggregate is incrementally updatable on the first insertion.

2. Induction hypothesis:

   Suppose for $n > 1$ that $\mathcal{F}(X_{n-1}, Y_{n-1}, \cdots, Z_{n-1})$ is incrementally updatable.[11]

3. Induction:

   On *n-th* insertion $(x_n, y_n, \cdots, z_n)$, by the induction hypothesis,

   $$\begin{aligned}\mathcal{F}(X_n, Y_n, \cdots, Z_n) &= (\sum_{i=1}^{n-1} f(x_i, y_i, \cdots, z_i, i)) + f(x_n, y_n, \cdots, z_n, n) \\ &= \mathcal{F}(X_{n-1}, Y_{n-1}, \cdots, Z_{n-1}) + f(x_n, y_n, \cdots, z_n, n).\end{aligned}$$

[11] It should be reminded that n is not directly used in aggregate computations. Its sole purpose is to distinguish the previous (or the next) update from the current update. If n appears in a summation body, it should be interpreted as aggregate *count(*)*.

Therefore, if $f(x_n, y_n, \cdots, z_n, n)$ can be incrementally computed, $\mathcal{F}(X_n, Y_n, \cdots, Z_n)$ should be incrementally updatable. By the postulation of the lemma, all entities in $f(x_n, y_n, \cdots, z_n, n)$ are bound, thereby their values, except those of bound component summative aggregates, are immediately known. Thus, if there is no component summative aggregate in $f()$, value of $f()$ can be computed immediately. Otherwise, $f()$ waits until values of all component aggregates, say, $\mathcal{F}'()$'s, become known. $\mathcal{F}'()$'s, in turn, are all bound by the postulation. Therefore, by applying induction steps similar to those thus far, $\mathcal{F}'()$'s can be immediately computed by adding its previous value to the value of its summation body, say, $f'(x_n, y_n, \cdots, z_n, n)$, if $f'(x_n, y_n, \cdots, z_n, n)$ can be computed immediately. Otherwise, $\mathcal{F}'()$ waits again. Applying these steps repeatedly, eventually computation will reach into the deepest nest where no summative aggregates are used. Then, summation body of that deepest aggregate can be computed by values in $(x_n, y_n, \cdots, z_n, n)$, and the recursion is unwound up into $f()$ and eventually $f(x_n, y_n, \cdots, z_n, n)$ is computed.

□

Theorem 4 *Every normalized summative aggregate is incrementally updatable.*

Proof of Theorem 4 Straightforward from Lemma 6 and the definition of normalization. □

Corollary 1 *Every normalized extended summative aggregate is incrementally updatable.*

Proof of Corollary 1 Straightforward from Lemma 5 and the definition of normalization of extended summative aggregates. □

It should be noted that Corollary 1 does not imply that the class of normalized (rather, normalizable) extended summative aggregates is the largest class of summative aggregates that can be incrementally updated. There are a few cases in which unnormalizable (by our normalization method) summative aggregates can be incrementally updated by "manually" normalizing them as shown in the following example.

Example 4.3.15 Given a summative aggregate, $\sum_{i=1}^{n} \log\left(\frac{n}{x_i}\right)$, if the user manually expands equation 4.21 into equation 4.22 (equivalently, if the user writes the summative aggregate in the form of equation 4.22), the given summative aggregate can be normalized to equation 4.24, thereby being incrementally updatable.

$$\sum_{i=1}^{n} \log\left(\frac{n}{x_i}\right) = \sum_{i=1}^{n} \log\left(\frac{\sum_{j=1}^{n} 1}{x_i}\right) \quad (4.21)$$

$$= \sum_{i=1}^{n} \left(\log \sum_{j=1}^{n} 1 - \log x_i\right) \quad (4.22)$$

$$= \sum_{i=1}^{n} \left(\log \sum_{j=1}^{n} 1\right) - \sum_{i=1}^{n} \log x_i \quad (4.23)$$

$$= \left(\log \sum_{j=1}^{n} 1\right) \sum_{i=1}^{n} 1 - \sum_{i=1}^{n} \log x_i. \quad (4.24)$$

$\triangleleft$

4.4 Looking-Up Cached Aggregates

In the preceding sections we have described how aggregates, especially summative aggregates, cached in the aggregate cache can be updated in an incremental way. In this section we investigate an efficient way of looking up a cached aggregate that is requested by a query.

Unlike in an ordinary cache mechanism, cache look-up has a great importance in the aggregate cache. First of all, cache look-up is no longer simple a task. For system built-in aggregates, finding them in the cache still remains simple since they must

have fixed names and fixed argument formats. However, for user-defined aggregates, their names cannot be good keys for locating them. Moreover, it is a doubtful approach to allow the users to name their aggregates. The aggregate cache needs to be as transparent to the users as possible. Explicitly naming an aggregate implies that if the name is forgotten or not known, the cached aggregate cannot be used or shared. Related and more important is the fact that even though an aggregate is first requested, thus not cached yet, it is possible that a similar aggregate is already cached and the requested aggregate can be derived from the cached one. If cache look-up is relied entirely on explicit naming, it is not possible to find out such a derivability.

Example 4.4.1 Suppose that two users are using the same database. Suppose further that *average* is not a built-in aggregate. One user wants to compute average over an aggregate input set X, and names it $avg(X)$. Not long after the first user has completed the computation, the second user wants the same aggregate but does not know that the same aggregate has been computed with a name *avg*. So, he/she requests the aggregate by a name $average(X)$. And, the same average is recomputed even though a copy is in the cache. Now, the third user comes in and wants *sum*, which is again supposed not to be built-in, over X. If $average(X)$ and $count(X)$ are in the cache, it should be possible to derive $sum(X)$ from them. But if any of the three is user-defined and its semantics is not correctly understood by the system, such a derivation would not be possible. $\lhd$

In our approach, we provide a small number of built-in (predefined) aggregates that have fixed names and semantics. For user-defined summative aggregates, they too can have names but those names are only for definitional convenience. That is, a user can define named summative aggregates and use the names in his/her queries to refer to the defined aggregates. However, when a summative aggregate

is cached, whether built-in or user-defined, it is decomposed and normalized into several smaller summative aggregates. Then, each of the decomposed summative aggregates is cached if that aggregate is not cached already. When a query requests a summative aggregate, the aggregate is looked up in the cache by following similar steps. The queried aggregate is parsed, decomposed and normalized, and each decomposed aggregate is searched in the cache. If all the decomposed aggregates are found in the cache, value of the queried aggregate is readily computed by using the information obtained when the aggregate is parsed and the values retrieved from the cache. Thus, for cached summative aggregates, their names are immaterial as long as cache look-up is concerned.

Example 4.4.2 This is a comprehensive example. When $S_{xy}(X_n, Y_n)$, which was shown in Example 4.3.1, is cached or looked up in the cache, the following equations

show how the original aggregate is decomposed and normalized.

$$
\begin{aligned}
S_{xy}(X_n, Y_n) &= \sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y}) && (4.25)\\
&= \sum_{i=1}^{n}(x_i - \frac{\sum_{j=1}^{n} x_j}{\sum_{k=1}^{n} 1})(y_i - \frac{\sum_{j=1}^{n} y_j}{\sum_{k=1}^{n} 1}) && (4.26)\\
&= \sum_{i=1}^{n}(x_i y_i - x_i\frac{\sum_{j=1}^{n} y_j}{\sum_{k=1}^{n} 1} - y_i\frac{\sum_{j=1}^{n} x_j}{\sum_{k=1}^{n} 1} + \frac{\sum_{j=1}^{n} x_j}{\sum_{k=1}^{n} 1}\frac{\sum_{j=1}^{n} y_j}{\sum_{k=1}^{n} 1}) && (4.27)\\
&= \sum_{i=1}^{n}(x_i y_i) - \sum_{i=1}^{n}(x_i\frac{\sum_{j=1}^{n} y_j}{\sum_{k=1}^{n} 1}) - \sum_{i=1}^{n}(y_i\frac{\sum_{j=1}^{n} x_j}{\sum_{k=1}^{n} 1})\\
&\qquad + \sum_{i=1}^{n}(\frac{\sum_{j=1}^{n} x_j}{\sum_{k=1}^{n} 1}\frac{\sum_{j=1}^{n} y_j}{\sum_{k=1}^{n} 1}) && (4.28)\\
&= \sum_{i=1}^{n}(x_i y_i) - \frac{\sum_{j=1}^{n} y_j}{\sum_{k=1}^{n} 1}\sum_{i=1}^{n} x_i - \frac{\sum_{j=1}^{n} x_j}{\sum_{k=1}^{n} 1}\sum_{i=1}^{n} y_i\\
&\qquad + \frac{\sum_{j=1}^{n} x_j}{\sum_{k=1}^{n} 1}\frac{\sum_{j=1}^{n} y_j}{\sum_{k=1}^{n} 1}\sum_{i=1}^{n} 1 && (4.29)
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{i=1}^{n}(x_i y_i) - \frac{\sum_{j=1}^{n} y_j}{n}\sum_{i=1}^{n} x_i - \frac{\sum_{j=1}^{n} x_j}{n}\sum_{i=1}^{n} y_i + \frac{\sum_{j=1}^{n} x_j}{n}\frac{\sum_{j=1}^{n} y_j}{n}n && (4.30)\\
&= \sum_{i=1}^{n}(x_i y_i) - \frac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n} && (4.31)
\end{aligned}
$$

First, $\overline{x}$ and $\overline{y}$ (averages of X_n and Y_n) in equation 4.25 is rewritten into summative aggregates, resulting in equation 4.26. Equation 4.26 is, then, expanded to equation 4.27, and $\sum$s are distributed over summation body of equation 4.27, resulting in equation 4.28. Now, the normalization is performed on equation 4.28. Equation 4.29 is a result of the normalization. It contains four unique normalized (decomposed) summative aggregates, $\sum_{i=1}^{n} x_i$, $\sum_{i=1}^{n} y_i$, $\sum_{i=1}^{n} x_i y_i$, and $\sum_{i=1}^{n} 1$ that should be incrementally updatable. These summative aggregates are first searched in the cache. If they are all found in the cache, then $S_{xy}(X_n, Y_n)$ can be obtained immediately. If any of them is not found, then that aggregate has to be recomputed from values in base databases (or in the data warehouse if an appropriate copy is being maintained). Value of $S_{xy}(X_n, Y_n)$ is obtained using cached/recomputed summative aggregates,

and all the recomputed summative aggregates are cached for later use. Later on, if other query requests any of the cached summative aggregates, say, summation of X_n, such an aggregate will be found in the cache until replaced out by other aggregate. On the other hand, equations 4.30 and 4.31 show a process of algebraic simplification of equation 4.29. However, we do not carry out any such simplification. In most cases, the number of unique summative aggregates does not decrease even after such a simplification. Therefore, penalty for not simplifying should be minimal. $\triangleleft$

4.5 Conclusions

In this work we have proposed the aggregate cache to improve performance of complex aggregate queries in the context of data warehouses. Considering aggregate computation is a frequent operation in data warehouse applications and such a computation is expensive to perform, reuse of once-computed results is a natural choice. On top of that, the temporal locality of aggregate accesses observed in decision making processes makes such a cache approach more attractive.

The Aggregate cache has a close bearing on the view materialization since a cached aggregate can be deemed as a materialized view of underlying tables in base databases. As the incremental view update is an important issue in the view materialization, incremental update of a cached aggregate as relevant underlying tables change is a crucial problem in the aggregate cache. If an underlying table is updated and the update affects a cached aggregate, the update should be propagated to the cache so that the cached aggregate too can be updated appropriately.

Based on currency requirement for cached aggregates, there are several schemes of when to update cached aggregates as base databases change. Rematerialization, periodic update, eager update, and on-demand update have been discussed, and the on-demand update has been chosen for the aggregate cache since it is not only perfectly geared with the cache philosophy, but also outperforms other approaches.

We have investigated a way of incrementally updating summative aggregates, which cover a vast variety of aggregates performing some types of cumulative operations. Importantly, we have identified a class of summative aggregates that is incrementally updatable and inclusive enough to cover many aggregates used in data warehouse applications, and proposed an efficient cache look-up method.

CHAPTER 5
CONCLUSIONS

In this work we have addressed two related issues within the framework of active databases. First, we have proposed a practical approach to static analysis of active rules and their confluent execution based on different user requirements. When the user wants the full confluent rule execution, that requirement can be easily met by removing conflicts in the rule set or by specifying priorities between the conflicting rules. If confluent rule execution is unnecessary, the system can avoid controlling rule executions. The confluence can also be enforced for only a subset of rules. Using our approach, it is also possible to support multiple, application-based confluency controls. In addition, our approach is the best fit for parallel rule execution.

In the second part of our work, we have proposed the aggregate cache that is a cache mechanism for aggregates used in data warehouses. The aggregate cache can improve the performance of aggregate computations significantly by saving previous results. It uses a novel approach to cache look-up, in which a queried aggregate is found in the aggregate cache by looking-up its component aggregates. Our aggregate cache is transparent to the user; no intervention from the user is necessary to run the aggregate cache. Also importantly, we have identified a precise class of aggregates that can be incrementally updated. We expect that the aggregate cache can be implemented by using active rules over active database systems. Change detection and propagation will be able to be implemented using the event detection facility in underlying active base databases. In the data warehouse – it is assumed to be an active database too, incrementally updating cached aggregates as changes are propagated to the aggregate cache can also be implemented using active rules. However,

the query processor of data warehouse should be modified appropriately to make use of cached aggregates.

REFERENCES

[1] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. Technical report, Department of Computer Science, Stanford University, Stanford, CA, 1995.

[2] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In *Proceedings International Conference on Very Large Data Bases*, pages 479–487, Barcelona, Spain, 1991.

[3] A. Aiken, J. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems*, 20(1):3–41, Mar. 1995.

[4] A. Aiken, J. Widom, and J. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings International Conference on Management of Data*, pages 59–68, San Diego, CA, 1992.

[5] E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proceedings International Conference on Management of Data*, pages 99–108, Washington, DC, May 1993.

[6] R. Badani. Nested transactions for concurrent execution of rules: Design and implementation. Master's thesis, CIS Department, University of Florida, Gainesville, FL, October 1993.

[7] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proceedings International Conference on Very Large Data Bases*, pages 475–486, Santiago, Chile, 1994.

[8] J. Blakely, P. Larson, and F. Tompa. Efficiently updating materialized views. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 61–71, Los Angeles, May 1986.

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

[10] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Australia, 1990.

[11] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, 1991.

[12] S. Chakravarthy, B. Blaustein, A. P. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, 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.

[13] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts, and detection. In *Proceedings International Conference on Very Large Data Bases*, pages 606–617, Santiago, Chile, Sep. 1994.

[14] S. Chakravarthy, Z. Tamizuddin, and J. Zhou. SIEVE: An interactive visualization and explanation tool for active databases. In *Proc. of the 2nd International Workshop on Rules in Database Systems (RIDS'95)*, pages 179–191, October 1995.

[15] O. Diaz, A. Jaime, and N. W. Paton. Dear: A debugger for active rules in an object-oriented context. In *Proc. of the 1st International Conference on Rules in Database Systems*, September 1993.

[16] K. P. Eswaran. Specifications, implementations, and interactions of a trigger subsystem in an integrated data base system. IBM Research Report RJ1820, Aug. 1976.

[17] S. Gatziu and K. R. Dittrich. SAMOS: An active, object-oriented database system. *IEEE Quarterly Bulletin on Data Engineering*, 15(1-4):23–26, December 1992.

[18] S. Gatziu and K. R. Dittrich. Events in an object-oriented database system. In *Proc. of the 1st International Conference on Rules in Database Systems*, September 1993.

[19] N. Gehani and H. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings International Conference on Very Large Data Bases*, pages 327–and 336, Barcelona, Spain, 1991.

[20] N. H. Gehani, H. V. Jagadish, and O. Shmueli. COMPOSE: A system for composite event specification and detection. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, December 1992.

[21] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an object-oriented database. In *Proceedings International Conference on Management of Data*, pages 81–90, San Diego, CA, June 1992.

[22] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings International Conference on Management of Data*, pages 328–339, San Jose, CA, 1995.

[23] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings International Conference on Very Large Data Bases*, pages 358–369, Zurich, Swizerland, 1995.

[24] A. Gupta, I. Mumick, and K. Ross. Adapting materialized views after redefinitions. In *Proceedings International Conference on Management of Data*, pages 211–222, San Jose, CA, 1995.

[25] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings International Conference on Management of Data*, pages 157–166, Washington, DC, 1993.

[26] E. Hanson. User-defined aggregates in the relational database system INGRES. Master's report, University of California, Berkeley, CA, 1984.

[27] E. Hanson. *Efficient Support for Rules and Derived Objects in Relational Database Systems*. PhD thesis, University of California, Berkeley, CA, 1987.

[28] E. Hanson. The design and implementation of the Ariel active database rule system. Technical Report UF-CIS-018-92, CIS Department, University of Florida, Gainesville, FL, 1992.

[29] E. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings International Conference on Management of Data*, pages 49–58, San Diego, CA, 1992.

[30] E. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(3):121–143, Sep. 1993.

[31] W. Inmon and R. Hackathorn. *Using the Data Warehouse*. John Wiley & Sons, Inc., New York, 1994.

[32] T. Sellis. Efficiently supporting procedures in relational database systems. In *Proceedings International Conference on Management of Data*, San Francisco, CA, 1987.

[33] IEEE Computer Society. IEEE data engineering bulletin: Special issue on materialized views and data warehousing, June 1995.

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

[35] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proceedings International Conference on Management of Data*, pages 281–290, Atlantic City, NJ, 1990.

[36] Z. Tamizuddin. Rule execution and visualization in active oodbms. Master's thesis, CIS Department, University of Florida, Gainesville, FL, May 1994.

[37] J. Widom. Research problems in data warehousing. In *Proceedings International Conference on Information and Knowledge Management (CIKM)*, Nov. 1995.

[38] J. Widom, R. Cochrane, and B. Lindsay. Implementing set-oriented production rules as an extension to starburst. In *Proceedings International Conference on Very Large Data Bases*, pages 275–285, Barcelona, Spain, 1991.

[39] J. Widom and S. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings International Conference on Management of Data*, pages 259–270, Atlantic City, NJ, 1990.

[40] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings International Conference on Management of Data*, pages 316–327, San Jose, CA, 1995.

BIOGRAPHICAL SKETCH

Seung-Kyum Kim was born on December 23, 1961, in Inju, Chungnam, South Korea. He received his Bachelor of Engineering degree in computer science from Ajou University, South Korea, in 1985. After his graduation, he worked as a research engineer at ETRI (Electronics and Telecommunication Research Institute) in South Korea. He fully participated in a research project developing a relational DBMS while working at ETRI.

In Fall 1990, he joined the Department of Computer and Information Science and Engineering at the University of Florida for his graduate studies. He received his Master of Science degree in computer and information science and engineering in 1993. Having continued his studies at the same department, he will receive his Doctor of Philosophy degree in May 1996.

His research interests include active databases, data warehouses, temporal databases, and transaction processing.