Active Rules based on Object Relational Queries

- Efficient Change Monitoring Techniques

by

Martin Sköld

Abstract

The role of databases is changing because of the many new applications that need database support. Applications in technical and scientific areas have a great need for data modelling and application-database cooperation. In an *active database* this is accomplished by introducing active rules that monitor changes in the database and that can interact with applications. Rules can also be used in databases for managing constraints over the data, support for management of long running transactions, and database authorization control.

This thesis presents work on tightly integrating active rules with a second generation Object-Oriented(OO) database system having transactions and a relationally complete OO query language. These systems have been named *Object Relational*. The rules are defined as Condition Action (CA) pairs that can be parameterized, overloaded, and generic. The condition part of a rule is defined as a declarative OO query and the action as procedural statements.

Rule condition monitoring must be efficient with respect to processor time and memory utilization. To meet these goals, a number of techniques have been developed for compilation and evaluation of rule conditions. The techniques permit efficient execution of *deferred rules*, i.e. rules whose executions are deferred until a *check phase* usually occurring when a transaction is committed.

A rule compiler generates *screener predicates* and *partially differentiated relations*. Screener predicates screen physical events as they are detected in order to efficiently capture those events that influence activated rules. Physical events that pass through screeners are accumulated. In the check phase the accumulated changes are incrementally propagated to the relations that they affect in order to determine whether some rule condition has changed. *Partial Differentiation* is defined formally as a way for the rule compiler to automatically generate partially differentiated relations. The techniques assume that the number of updates in a transaction is small and therefore usually only some of the partially differentiated relations need to be evaluated. The techniques do not assume permanent materializations, but this can be added as an optimization option. Cost based optimization techniques are utilized for both screener predicates and partially differentiated relations. The thesis introduces a calculus for incremental evaluation based on partial differentiation. It also presents a propagation algorithm based on the calculus and a performance study that verifies the efficiency of the algorithm.

Contents

Preface

This thesis presents work in two areas of active database research. First, it presents work on integrating active rules into an Object Relational Database System(ORDBMS) called AMOS [35]. Secondly, it presents work on efficient change monitoring of rule conditions. These two parts are fairly unrelated. The first part considers the extension of the data model of AMOS with rules which is a matter of rule expressability. The rule model presented here can be introduced into any ORDBMS.

The second part considers the efficiency of rule execution which is a matter of performance. The techniques that are presented for efficient rule condition monitoring are general and can be used in any active database system.

The two parts are, however, not completely unrelated. The rules that are presented are based on the idea that the user should not have to specify any procedural information of how the rule condition is to be monitored. This information should be deduced by the database. This requires that the database can efficiently monitor any complex rule condition that the user defines.

Thesis Outline

Chapter 1 introduces the work done on integrating active rules into AMOS and the techniques that have been developed for efficient change monitoring of rule conditions.

Chapter 2 introduces the research area of active databases and the AMOS architecture.

Chapter 3 defines the data model of AMOS, the query language AMOSQL, and the extension of AMOSQL with rules. Examples are also given that further explain how the rules can be used.

Chapter 4 defines the semantics of AMOSQL rules and how condition monitoring is related to function monitoring. The techniques of generating screener predicates and partial Δ-relations are introduced.

Chapter 5 defines the theoretical foundation for the incremental evaluation by specifying a calculus based on changes and by evaluating partial Δ-relations.

Chapter 6 discusses how rules are related to the transactions in which they are created, deleted, activated, deactivated, triggered, and executed. How rules can be used for transaction management is also discussed. Chapter 6 ends with a discussion on how the update semantics of the database affects the propagation algorithm described in chapter 8.

Chapter 7 discusses how query optimization techniques can be enhanced for optimization of screener predicates and partial Δ-relations.

Chapter 8 outlines the algorithm used to implement the incremental evaluation of rule conditions. The algorithm performs a bottom-up, breadth-first propagation of changes through a propagation network.

Chapter 9 compares the efficiency of the incremental method with the naive method based on experiments.

Chapter 10 concludes with a summary of the presented techniques and future work.

Financial Support

This work has been supported by NUTEK (The Swedish National Board for Industrial and Technical Development, TFR (The Swedish Technical Research Council), and CENIT (The Center for Industrial Information Technology), Linköping University.

Acknowledgements

I would like to thank Professor Tore Risch for his continuous support and for introducing me into the area of active databases. Tore brought the WS-Iris system with him from HP-labs and his system has now become the AMOS system. Because of this the research project got a running start.

I would also like thank all the other members of the lab for Engineering Databases and Systems (EDSLAB) for inspiration and for fruitful discussions on the development of the AMOS system.

Martin Sköld

Linköping, June 1994

1 Introduction

1.1 Background and Orientation

The role of databases is changing because of the many new applications that need database support. Applications in technical and scientific areas have a great need for data modelling and application-database cooperation.

The limitations of relational databases when it comes to data modelling has led to the development of new database technology based on Object Oriented techniques. In the first generation of Object Oriented (OO) databases the systems were built by adding persistency to OO programming languages. The query languages in these systems were limited to procedural iterators over data. The second generation of OO databases, called *Object Relational* Database Systems(ORDBMS), will include relationally complete query languages. Such systems are already emerging and will probably be based on standards for OO extensions of relational query languages such as SQL-3[7]. The next generation databases, both relational and OO, will also include extended capabilities for constraint management, event triggering, and database-application interaction.

The cooperation between the database and applications can consist of monitoring specific changes in the database that are of interest to an application. *Active databases* provide applications with the possibility of specifying rules that monitor changes in the database that inform the applications of interesting changes. The need for data modelling also includes the need for specifying constraints over the data in order to enforce the integrity of the data for an application. In an active database these integrity constraints can be specified as constraint rules that monitor changes that might violate a constraint. The constraint rules can undo these changes either by providing compensating updates that restores the integrity of the data or by aborting the transaction that performed the changes.

1.2 Summary of Contributions

This thesis presents work done on integrating active rules into an Object Relational Database System(ORDBMS) and work on efficient change monitoring of rule conditions.

1.2.1 Introducing Active Rules into an ORDBMS

Active rules have been introduced into the AMOS[35] ORDBMS which is further described in the thesis. The rules are integrated into AMOSQL, the query language of AMOS. The rules are of CA (Condition Action) type, where the Condition is an AMOSQL query and the Action can be any sequence of AMOSQL procedure statements. Rules monitor changes to the rule conditions and data can be passed from the Condition to the Action of each rule by using shared query variables, i.e. set-oriented Action execution[72] is supported. By modelling rules as objects it is possible to make queries over rules. Overloaded and generic rules are also allowed, i.e. rules that are parameterized and can be activated for different types.

1.2.2 Efficient Change Monitoring Techniques

As mentioned above, the ability to perform change monitoring is introduced by rules in active databases. When doing change monitoring in a database it is crucial that the overall performance of the database is not impaired to any great extent. Rule monitoring is the activity of monitoring changes of the truth value of rule conditions. A *naive* method of detecting changes is to execute the complete condition of a rule. This, however, can be very costly, since a rule condition can span over large portions of the database.

Rule condition monitoring must not decrease the overall performance to any great extent, with respect to either processor time or memory utilization. The following techniques for compilation and evaluation of rule conditions have been developed to meet these goals:

- To efficiently determine changes to all activated rule conditions, given updates of stored data, a *rule compiler* analyses rule conditions and generates change detection plans.
- To minimize unnecessary execution of the plans, *screener predicates* that screen out uninteresting changes are generated along with the change detection plans. The screener predicates are optimized using cost based query optimization techniques.
- For efficient monitoring of rule conditions, the rule compiler generates several *partially differentiated relations* that detect changes to a derived relation given changes to one of the relations it is derived from. The technique is based on the assumption that the number of updates in a transaction is usually small and therefore only small effects on rule conditions will occur. Thus, the changes will only affect some of the partially differentiated relations. The partially differentiated relations are optimized using cost based query optimization techniques.
- To efficiently compute the changes of a rule condition based on changes of subconditions, the partially differentiated relations are computed by *incremental evaluation* techniques [9] [59].
- To correctly and efficiently propagate both insertions and deletions (positive and negative changes) without unnecessary materialization or computation, the calculation of changes to a relation must be preceded by the calculation of the changes to all its sub-relations. This is accomplished by a *breadth-first, bottom-up* propa-

gation algorithm, which also ensures graceful degradation as the complexity of rule conditions and as the size of the database increases.

Incremental evaluation techniques are based on using incremental changes as bases for evaluation instead of evaluating the full expressions. A good analogy is that of *spreadsheet* programs. Take a simple example of a spreadsheet table consisting of three columns A, B, and C (A+B), see fig. 1.1. In the last cell in each column the sum of the cells above is stored. If one cell of column A or B is changed then the sum A+B of that row will have to be recalculated. The other rows do not have to be checked since they have not changed. This is basically the idea behind incremental change monitoring of rule conditions. Rule conditions can be seen as equations that we want to monitor in order to determine if the rule should be triggered by some specific change. The conditions can, however, reference data in many different tables in one equation. The tables represent different database relations.

The total sum for each column in the spreadsheet example will have to be recalculated as well. By using the difference between the new and the old value the recalculation can be done efficiently. This is how incremental change monitoring of aggregation functions is done, see section 5.4.

	A	B	C
0	200	600	A0+B0 = 800
1	300	700	A1+B1 = 1000
2	400	800	A2+B2 = 1200
3	500	900	A3+B3 = 1400
4	ΣA = 1400	ΣB = 3000	ΣC = 4400

Figure 1.1: A spreadsheet example

The rule compiler analyses the execution plan for the condition of each rule and determines what functions the condition depends on. The output of the rule compiler is a plan for determining changes to all activated rule conditions, given updates of stored functions. The rule processor uses incremental evaluation techniques for efficiently computing the changes of a derived function based on changes of sub-functions. The compiler generates Δ-*relations* that for given updates represent all the net changes of a relation which a rule condition depends on. The Δ-relations are defined in terms of several *partial* Δ-*relations* that efficiently computes the changes of a derived function based on changes of a *single* sub-function. This is called *Partial Differentiation* of derived functions. The technique assumes that the number of updates in a transaction is small and therefore usually only small effects on rule conditions will occur. Thus, the changes only affect some of the partial Δ-relations. For updates that have large effects on the rule conditions the rule evaluation will have to be complemented with other techniques to be efficient, e.g. full evaluation of rule conditions or view materialization techniques[9] to re-use partial results.

Partial differentiation will be defined formally as a way to automatically generate Δ-relations from *CA-rules* (Condition-Action rules). A *Δ-set* is defined as a 'wave-front' materialization of a Δ-relation that exists temporarily and is cleared as the propagation proceeds upwards. The operator *delta-union* ($\cup_{\Delta}$) is defined to calculate a Δ-set from incremental changes. For good memory utilization, the technique avoids permanent materialization of large intermediate relations that span over a large number of objects. Such materialized relations can be very large and can even be considerably larger than the original database, e.g. where Cartesian products or unions are used. When many conditions are monitored and the database is large, complete materialization will become infeasible; thus the database will not scale up.

By using incremental evaluation techniques for rule condition execution the cost of rule condition monitoring can be reduced significantly. There have been significant work done in outlining algebras for incremental evaluation, but the actual algorithms and how they relate to other database functionality is not outlined in any great detail. Areas that affect these algorithms include transaction management, update semantics, materialization, and query optimization. This thesis introduces a calculus for incremental evaluation of rule conditions as well as a propagation algorithm for propagating changes. The more specific topics include a calculus for incremental evaluation of queries based on *partial differencing*, transactional management of rule creation/deletion and of the network for rule activation/deactivation, avoidance of unnecessary materialization, effects of different update semantics on the propagation algorithm, and query optimization techniques for enhancing performance, an algorithm for incremental evaluation based on breadth-first propagation of changes in a network, and a performance study of the incremental algorithm.

1.3 Related Work

The pioneering work done in introducing rules into databases was carried out in the HiPAC project [16][27]. In the project different rule semantics were defined. The system was, however, not implemented in full. Rule systems were implemented in POSTGRES[69] and Starburst[53]. In Ariel[41] CA-rules were introduced that resembled the CA-rules in AMOS. In Ode[39] active capability was introduced to an OODBMS. In section 2.3 more information about these systems can be found as well as other related work.

In [68] a relational approach is taken on the monitoring of complex systems. In [62] a model for functional monitoring of objects in an OODBMS is presented. This model of functional monitoring is adopted and extended in the integration of rules into AMOS.

General work on incremental evaluation can be found in [9][59]. Theoretical work on incremental evaluation of queries can be found in [6][60]. Related work on propagation of changes in production systems can be found in [55]. Directly related work on incremental change monitoring techniques can be found in [30][34][41][43][47].

For more detailed discussions of how different work relate to the work presented in this thesis, see related work at the end of each chapter.

2 Active Databases

2.1 Active versus Passive Databases

Traditional databases are *passive* in the sense that they are explicitly and synchronously invoked by user or application program initiated operations. Applications send requests for operations to be performed by the database and wait for the database to confirm and return any possible answers. The operations can be definitions and updates of the schema, as well as queries and updates of the data. An *active* database can be invoked, not only by synchronous events that can have been generated by users or application programs, but also by external asynchronous events such as changes of sensors or time. When monitoring events in a passive database a *polling technique* or *operation filtering* can be used to determine changes to data. With the polling method the application program periodically polls the database by placing a query about the monitored data. The problem with this approach is that the polling has to be fine tuned as not to flood the database with too frequent queries that mostly returns the same answers, or in the case of too infrequent polling, the application might miss important changes of data. Operation filtering is based on that all change operations sent to the database are filtered by an application layer that does the situation monitoring before sending the operations to the database. The problem with this approach is that it greatly limits the way condition evaluation can be optimized. It is desirable to be able to specify the conditions to monitor in the query language of the database. By checking the conditions outside the database the complete queries representing the conditions will have to be sent to the database. Many database systems allow precompiled procedures that can update the database. The effects of calling such a procedure cannot be determined outside of the database.

If the condition monitoring is used to determine inconsistencies in the database, it is questionable whether this should be performed by the applications, instead of the database itself. In an active database the condition monitoring is integrated into the database. This makes it possible to efficiently monitor conditions and to notify applications when an event occurred that caused a condition to become true and that is of interest to the application. Monitoring of specific conditions represented as database queries can be done more efficiently since the database have more control of how to evaluate the condition efficiently based on knowledge of what has changed in the database since the condition was last checked. It also lets the database perform consistency maintenance as an integrated part of the data management.

Internal database functions that can use data monitoring includes, for example, constraint management, management of long-running transactions, and authorization control. In constraint management rules can monitor and detect inconsistent updates and abort any transactions that violate the constraints. In some cases compensating actions can be performed to avoid inconsistencies instead of performing a roll-back of the complete transaction. In management of long-running transaction rules can be used to efficiently determine synchronization points of different activities and if one transaction has performed updates that have interfered with another [28]. This can be used, for example, in cooperation with *sagas*[37] where sequences of committed transactions are chained together with information on how to execute compensating transactions in case of a saga roll-back. In authorization control rules can be used to check that the user or application has permission to do specific updates or schema changes in the database.

Applications which depend on data monitoring activities such as CIM[1][52], Medical[44] and Financial Decision Support Systems[20] can greatly benefit from integration with databases that have active capabilities.

2.2 Active Databases and other Rule Based Systems

At a first glance it might seem that active databases are in some sense similar to *knowledge based systems*[45] and in other senses to *reactive systems*[54]. There are, however, some fundamental differences. An active database has basic database functionality such as transactions and a query language that give consistent and declarative access to data. The rules provide a handle to *monitor*[12] changes in the database. The database can detect changes of data by monitoring changes to rule conditions that express specific situations, or database states that are of interest. Active databases are only partly rule driven and most changes are not side-effects of other rules. In active databases there is a clear separation between the condition of a rule and the events that causes the condition to be evaluated. The possibility of modelling complex events is considered equally as important as modelling complex conditions.

In knowledge based systems the rules are used for *reasoning* using facts in a knowledge base. In these systems there is usually no clear distinction between events and rule conditions. Knowledge based systems usually provide different kinds of rules such as both forward and backward chaining rules and usually also provide more control of the rule inference machine. The rules can be used to build *Theorem Provers*[57] and *Truth Maintenance Systems* (TMS)[31]. These systems are often used to model complex behaviour, often based on uncertainties, through a large number of rules over a fairly limited amount of data. Support for grouping rules and explanatory functions that explains ‘why’ the system behaved in a certain way are common in this systems. In active databases the number of rules is usually smaller than in knowledge based systems,

1. Computer Integrated Manufacturing

but the amount of data that the rules are defined over is usually large, sometimes very large.

In reactive systems the rules are used for *control* of a physical environment. These rules are usually event driven with no conditions or fairly uncomplicated conditions. There is usually no database at all, all events come from changes in the physical environment. The rules that trigger usually directly control something in the physical environment which in turn generate events that again trigger some rule and so on. These kind of systems are usually real-time systems with a concept of time and a high degree of parallelism.

In reality, of course, there are no pure active, knowledge based or reactive sys-

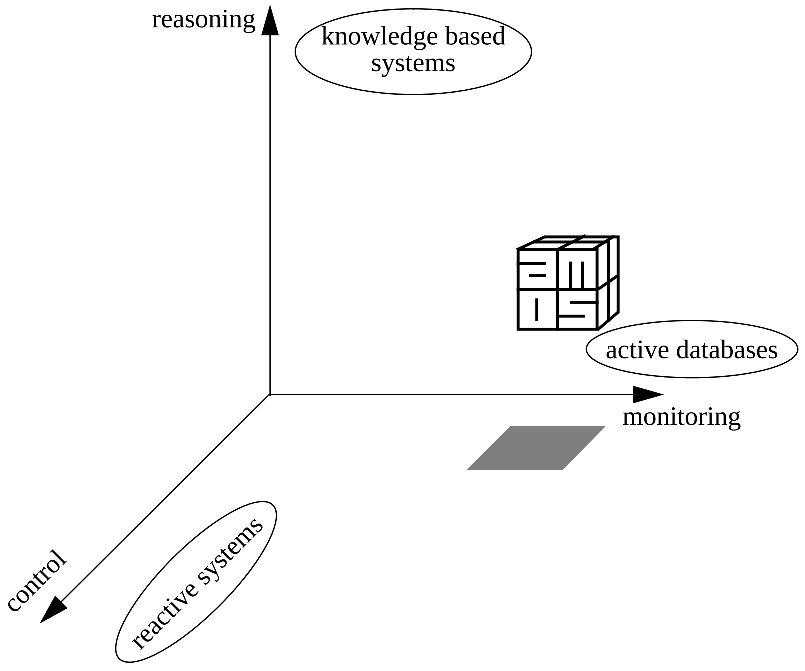


Figure 2.1: The relation between active databases and other rule based systems

tems, all rule based systems incorporate some monitoring, reasoning and control(fig. 2.1). There are, however, differences between how much of these can be found in a particular system. By mapping external events, that signal changes in a physical environment, into an active database [24], the system becomes partly reactive. The same can be done with a knowledge based system, as is done in *real-time knowledge based systems*[50]. Active databases that provide advanced constraint reasoning capabilities such as [13], or self reflective rules as in [33], can be seen as moving from active databases closer to knowledge based systems. *Demons* and *blackboard* based systems[32] can be seen as moving from knowledge based systems towards active databases. Introducing complex sensors and *sensor fusion* techniques [22] in reactive systems, can be seen as moving closer to active databases, since the rules now trigger on more complex events or conditions and the state of the sensors is usually saved in some simple database. As can be seen in fig. 2.1, AMOS is mainly based on monitoring, but can also be seen as having limited reasoning and control capabilities.The reasoning in AMOS is based on having the declarativeness of AMOSQL queries in the rule conditions. The control in rule actions is limited to updating the database or by calling applications that in turn control some external environment. The architecture of AMOS is presented in section 2.5.

In some system architectures, the reasoning, the monitoring and the control are seen as different layers of the architecture [52].

2.3 Active Databases, a Short Survey

In System R [3] a *trigger* mechanism was defined that could execute a prespecified sequence of SQL statements whenever some triggering event occurred. The triggering events that could be specified included retrieval, insertion, deletion and update of a particular base table or view. Triggers have immediate semantics, i.e. they are executed immediately when the event is detected. In System R *assertions* were also possible that specify permissible *states* or *transitions* in the database through integrity constraints that always have to be true after each transaction. Specific events have to be specified for when assertions are to be checked as with triggers. Assertions have deferred checking semantics, i.e. they are usually checked when transactions are to be committed.

The term *active databases* was coined by [56] as “a paradigm that combines aspects of both database and artificial intelligence technologies”. In [56] a mechanism for constraint maintenance, *Constraint Equations*, was presented as a declarative representation for a set of related Condition-Action rules.

In HiPAC [16][27] a thorough specification was done of what different mechanisms were desirable in an active database system. Rules are defined as Event-Condition-Action (ECA) rules, where the Event specifies when a rule should be triggered, the Condition is a query that is evaluated when the Event occurs, and the Action is executed when the Event occurs and the Condition is satisfied. In HiPAC *coupling modes*(fig. 2.2) were defined which specified how the evaluation of rule conditions and the execution of rule actions were related to the detected events and the transaction in which the events occurred. *Immediate* rule processing means that the rule conditions are evaluated and the actions are executed immediately after the event occurred. A separation was also made between if the rule processing takes place before or after the update has taken place in the database. *Deferred* rule processing means that rule processing is delayed until the transaction is to be committed. *Casually Dependent Decoupled* rule processing means that any triggered action execution is executed in a separate sub-transaction that waits until the main transaction is committed. Decoupled rule processing means that the sub-transaction is completely decoupled from the main transaction and commits regardless of the outcome of the main transaction.

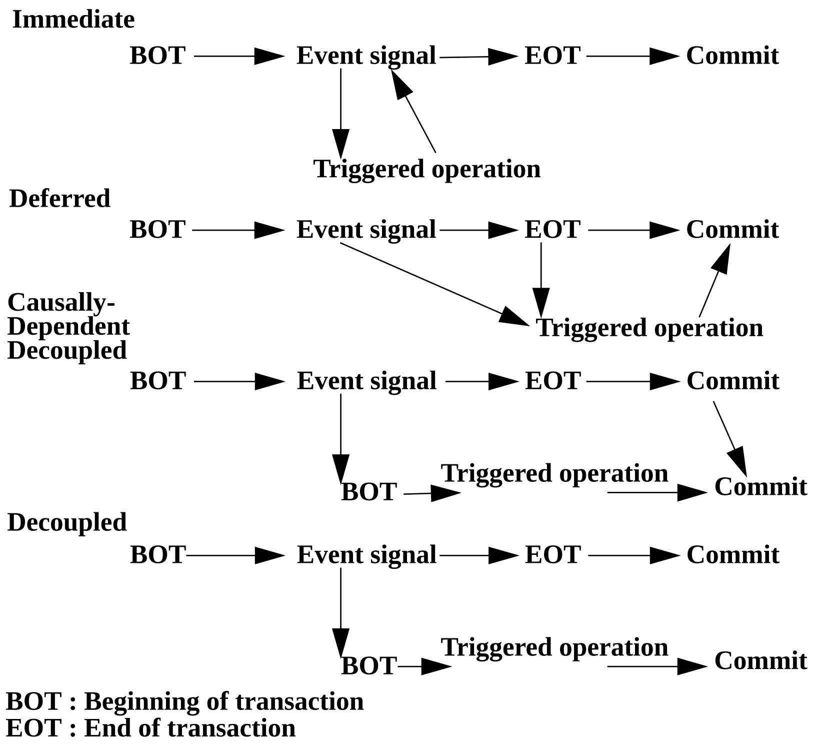


Figure 2.2: Rule processing coupling modes in HiPAC

In POSTGRES [69] rules were introduced as ECA rules where events can be *retrieve*, *replace*, *delete*, *append*, *new* (i.e replace or append), and *old* (i.e. delete or replace) of an object (a relation name or a relation column). The condition can be any POSTQUEL query and the action any sequence of POSTQUEL commands. Two types of rule systems exists, the *Tuple Level Rule System* which is called when individual tuples are updated, and the *Query Rewrite System* which resides in the parser and the query optimizer. The Query Rewrite System converts a user command to an alternative form which checks the rules more efficiently. No support exists for handling temporal, external events, and composite events.

In Starburst [53] ECA rules were introduced and the events can be INSERT, DELETE, and UPDATE of a table. The condition can be any SQL query and the action any sequence of database commands. Rules that are defined can be temporarily deactivated and then be re-activated. The condition and action parts may refer to *transition tables* that contain the changes to a rule's table made since the beginning of the transaction or the last time that a rule was processed (whichever happened most recently). The transition table INSERTED/ DELETED contains records inserted/deleted into/from the trigger table. Transition tables NEW_UPDATED and OLD_UPDATED contain new and old values of updated rows, respectively. In [72] the *set-oriented* semantics of Starburst rules is presented. In a set-oriented rule the action part is executed for all tuples for which the condition is true.

Other systems based on ECA-rules are [11][38].

In Ariel [41] production rules were defined on top of POSTGRES. In Ariel CA-rules were allowed which use only the condition to specify *logical events* which trigger rules.

In Ode [39] constraints and triggers were introduced into an Object Oriented database. The *basic events* that can be referenced are creation, deletion, update, or access by an object method. Ode also supports *composite events* through event expressions that relate basic events. The event expressions can define sequence orderings between events.

In both POSTGRES[69] and Starburst[53] events are intercepted in a similar manner as in AMOS. However, the events that are intercepted in AMOS include all operations of high-level objects. This makes it possible to extend rules to trigger on any change in the system, including schema updates. This is further discussed in section 3.2.

Systems that can trigger on external events include [11][38].

2.4 Active Database Classifications

Considerable research has been carried out in the area of active databases. There exist several good introductory papers to active database architectures [19][42]. Two important evaluation aspects for comparing different architectures are the expressiveness of the rule language and the execution semantics of the rules.

The expressiveness of the rules can be divided into the expressiveness of rule events, conditions and actions. The expressiveness of the event part can be divided into comparing the types of events the rules can reference and how the events can be modelled and combined into complex events. Different types of events include database updates, schema changes and external events such as sensor changes, specified state changes in the applications, or time. Modelling events can include an event specification language that can combine events using logical composition, event ordering, sequential and temporal ordering, and event periodicity [17].

The expressiveness of the condition part can be divided into whether a full query language is available or not, if events can be referenced as changed data and if old values can be referenced or not.

The expressiveness of the action part can be divided into whether a full query language is available or not, i.e. if queries and updates can be intertwined, and can include schema changes and rule activation/deactivation.

Execution semantics of rules includes rule processing coupling modes defined in section 2.3. If full query language expressiveness is possible in the condition part, then set-oriented rule semantics is also possible [72], where the action part is executed over a set of tuples produced by the condition. Cascading rule execution, i.e. whether one rule can trigger another, and if simultaneously triggered rules are subjected to some conflict resolution method are also part of the classification of rule semantics.

2.5 AMOS

AMOS[35] (Active Mediators Object System) is an architecture to model, locate, search, combine, and monitor data in information systems with many workstations connected using fast communication networks. The architecture uses the *mediator* approach [73] that introduces an intermediate level of software between databases and their use in applications and by users. We call our class of intermediate modules *active mediators*, since our mediators support active database facilities. The AMOS architecture is built around a main memory based platform for intercommunicating information bases. Each AMOS server has DBMS facilities, such as a local database, a data dictionary, a query processor, transaction processing, and remote access to databases. AMOS is an extension of a main-memory version of Iris[36], called WS-Iris[51], where OSQL queries are compiled into execution plans in an OO logical language called ObjectLog[51]. The query language of AMOS, AMOSQL, is a derivative of OSQL. AMOSQL extends OSQL with active rules, a richer type system and multi-database functionality. In the development of AMOSQL there is also an ambition to adapt to the future SQL-3[7] standard, but with the extensions mentioned above.

The AMOS architecture (fig. 2.3) is a layered architecture consisting of seven levels.

- The *external interface* level can handle synchronous requests through a client-server interface for loosely coupled applications and through a fast-path interface for tightly coupled applications. The interface also handles asynchronous interrupts as well as database-application call-backs. All synchronous interaction is done through the AMOSQL interface. Asynchronous interrupts that signal external events such as timer events or changes to external sensors are transformed into database events and sent to the event manager.

- The *AMOSQL interface* parses AMOSQL expressions and sends requests to the levels below. A fast path interface that does not require any parsing is also available. Any results are returned to the external interface, either directly or through interface variables and cursors.

- The *event manager* dispatches events to the rule processor. Events can come either from the external interface or from intercepted events in lower levels such as schema updates or relational updates.

- The *schema manager* handles all schema operations such as creating or deleting types, i.e. object classes, and type instances including functions and rules. The *query processor* handles query optimization and query execution.

- The *rule processor* handles compilation, activation, monitoring and execution of rules and is further described below.
- The *high level object manager* manages all operations to all objects in the database schema such as object creation, deletion and updates of object attributes including updating, inserting and deleting data, in stored functions, i.e. base relations. The level also handles OIDs (Object Identifiers) of the objects. All operations on these objects are transactional and are thus logged. All operations generate events that are intercepted and sent to the event manager.
- The *transaction manager* handles all database transactions by keeping an undo/redo log of all database operations.
- The *recovery manager* ensures persistency by making periodical snapshots and flushing the log to disk.
- The *low level object manager* handles all basic objects (everything in the database is an object) such as lists, vectors, hash tables, atoms, strings, integers and reals.
- The *memory manager* manages all memory operations such as allocation, deallocation and garbage collection.

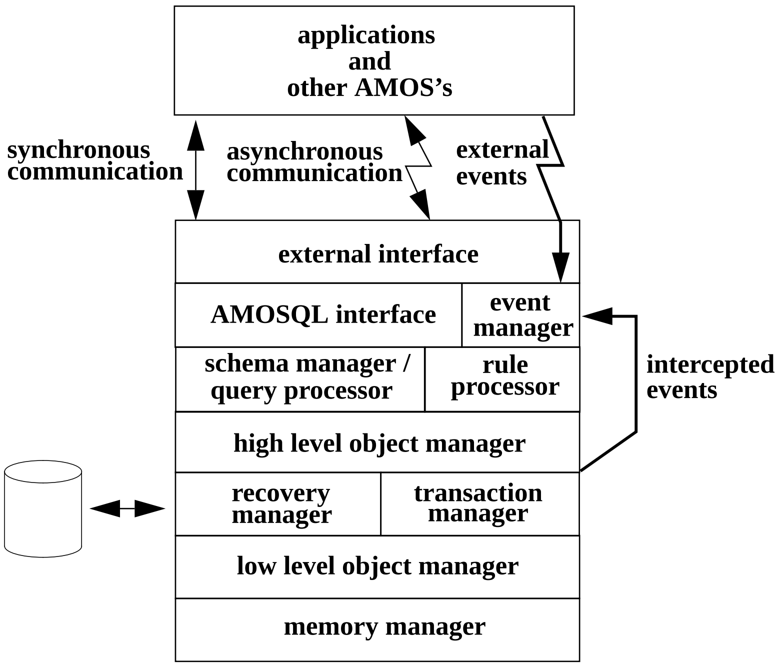


Figure 2.3: The AMOS architecture

The event handling is tightly integrated into the system and internal changes are intercepted where they occur in the lower levels for efficiency reasons. The rule processor is tightly integrated with the query processor for the same reason.

2.6 The Rule Processor

The rule processor handles rule creation/deletion, activation/deactivation, monitoring, and execution. The processing of rules is divided into four phases:

1. Event Detection
2. Change monitoring
3. Conflict resolution[1]
4. Action execution

Event detection consist of detecting events that can affect any activated rules and is performed continuously during ongoing transactions. Change monitoring includes using the detected events to determine if any condition of any activated rules have changed, i.e. have become true. During action execution further events might be generated causing all the phases to be repeated until no more events are detected. Different conflict resolution methods are outside the scope of the thesis. In the current implementation a simple priority based conflict resolution is used.

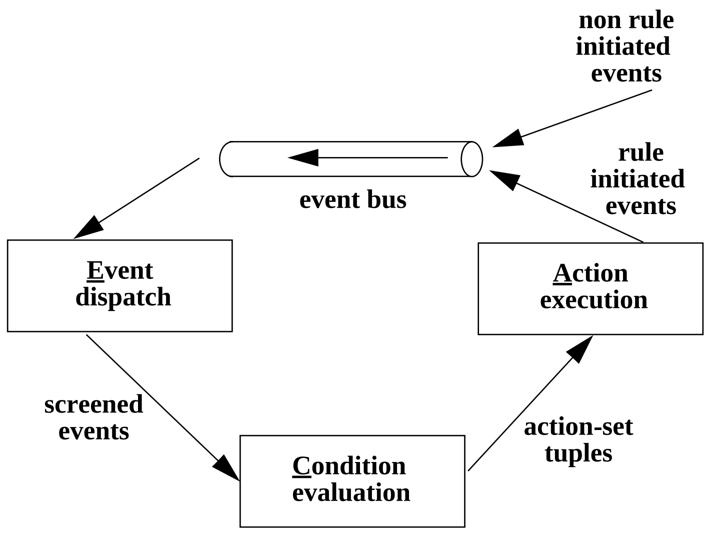


Figure 2.4: The ECA execution cycle

1. Conflict resolution is the process of choosing one single rule when more than one rule is triggered.

The rule execution model in AMOS is based on the *Event Condition Action* (ECA) execution cycle (fig. 2.4).

All events are sent on a software bus, i.e. an event queue, called the *event bus*. The execution cycle is always initiated by non rule initiated events such as database updates, schema changes, time events, or other external events. All events are dispatched through table driven execution. A screening is made of events that might change the truth values of rules. Rule conditions are evaluated based on the screened events to produce *action-sets* that contain tuples for which the actions are to be executed. When the actions are executed new events might be generated and the execution cycle continues until no more events are detected on the bus.

The rules in AMOS are of Condition Action (CA) type where the involved Events are calculated from the Condition by the rule compiler. The rules can be classified according to the aspects presented section 2.4. The expressiveness of events is planned to have all the full expressiveness of the derived functions in AMOSQL, i.e. full logical composition, as well as having the possibility of expressing event ordering and periodicity. Temporal event specifications are also considered. The expressiveness of conditions is based on the availability of complete AMOSQL queries in the condition. The expressiveness of actions is based on full AMOSQL procedural statements, i.e. queries intertwined with any updates of the schema, updates of functions, rule activation/deactivation, and application call-backs. The rules in the current implementation are only deferred, but immediate rules are planned.

3 Object Relational Query Rules

3.1 The Iris Data Model and OSQL

The data model of AMOS and AMOSQL are based on the data model of Iris and OSQL[36]. The Iris data model is based on objects, types and functions (fig. 3.1).

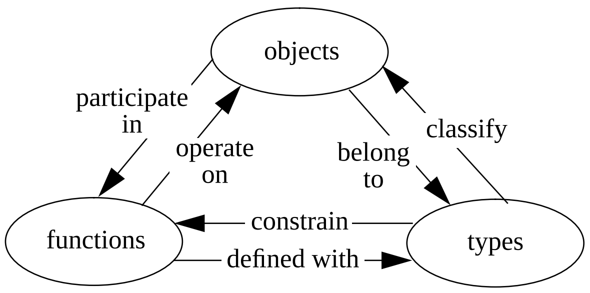


Figure 3.1: The Iris data model

Everything in the data model is an object, including types and functions. All objects are classified by belonging to one or several types, which equals object classes. Types themselves are of the type ‘type’ and functions are of the type ‘function’.

The data model in Iris is accessed and manipulated through OSQL[1]. All examples of actual schema definitions and database queries will here be written in a courier font.

For example, it is possible to define user types and subtypes:

```
create type person;
create type student subtype of person;
create type teacher subtype of person;
create type course;
```

1. The OSQL presented here is the WS-Iris dialect, which differs slightly from the OSQL in Iris and subsequent commercial products.

Stored functions can be defined on types that equals attributes in Object Oriented database or base relations in Relational databases, hence we call this model *Object Relational*. One function in the Iris data model equals several functions in a mathematical sense.

For example, a function can both give the name of a person given the person object or give all the person objects associated with a name.

```
create function name(person) -> charstring as stored;
```

Stored functions is the default:

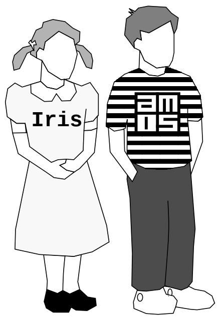
```
create function studies(student) -> course;
create function gives(teacher) -> course;
```

Derived functions equals methods or relational views and can be defined in terms of stored functions (and other derived functions).

```
create function teaches(teacher t) -> student s
       as select s for each course c where
       gives(t) = c and
       c = studies(s);
```

Instance objects of a type can be created and stored functions can be set for these instances:

```
create student instances :iris[1], :amos;
set name(:iris) = “Iris”;
set name(:amos) = “AMOS”;
create course
       instances :active_databases;
set studies(:amos) = :active_databases;
```



Multiple types (multiple inheritance) is possible by adding more types to an object:

```
add teacher to :amos;
```

Procedures are defined as functions that have side-effects:

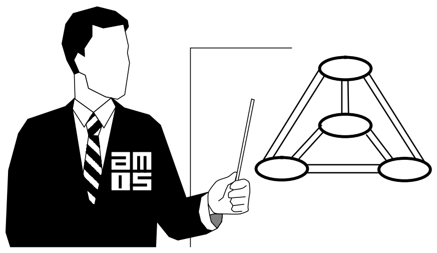
```
create function teach(teacher, student, course)
       -> boolean[2]
       as begin
            set gives(teacher) = course;
            set studies(student) = course;
          end;
```

1. These are interface variables and are not part of the database.
2. A procedure that does not explicitly return anything implicitly return a boolean.

Procedures are called by:

```
call teach(:amos, :iris, :active_databases);
select name(t)
  for each teacher t
  where teaches(t) = :iris;

<“AMOS”>
```



In the previous example the last query returns a single tuple. Queries, and subsequently functions, can return several tuples. Duplicate tuples are removed from stored functions if they are not explicitly defined to return a bag. We say that we have *set-oriented semantics*. *Bag-oriented semantics* is available as an option and can be specified along with the return type of a function.

Functions can be overloaded on the types of their arguments, i.e. we can define the same function in several ways depending on the types of the arguments. The system will in most cases choose the correct function at compile time, we call this *early binding*. In some cases the system can not determine what function to choose at compile time and must check some types at run time, we call this *late binding*. Since types and functions are objects as well, with the types ‘type’ and ‘function’, it is possible to define generic functions, i.e. functions that take types as arguments, and higher order functions, i.e. functions that take other functions as argument.

A transaction is aborted and rolled back by:

```
rollback;
```

A transaction can be finished and made permanent by:

```
commit;
```

3.2 The AMOS Data Model and AMOSQL

The AMOS data model extends that of Iris by introducing rules (fig. 3.2). Rules are also objects[26] and of the type ‘rule’. Rules monitor changes to functions and changes to functions can trigger rules. All the events that the rules can trigger on are modelled as changes to values of functions. This gives us the power of AMOSQL functional expressions as our event modelling language. Functions are seen as having passive (synchronous) or active (asynchronous) behaviour depending on if they are used in a query or in a rule condition. Passive functions display synchronous polling behaviour while active functions display asynchronous interrupt behaviour. Purely passive functions are functions that never changes, such as built in arithmetic functions, e.g. `+`, `-`, `*` and `/`, boolean

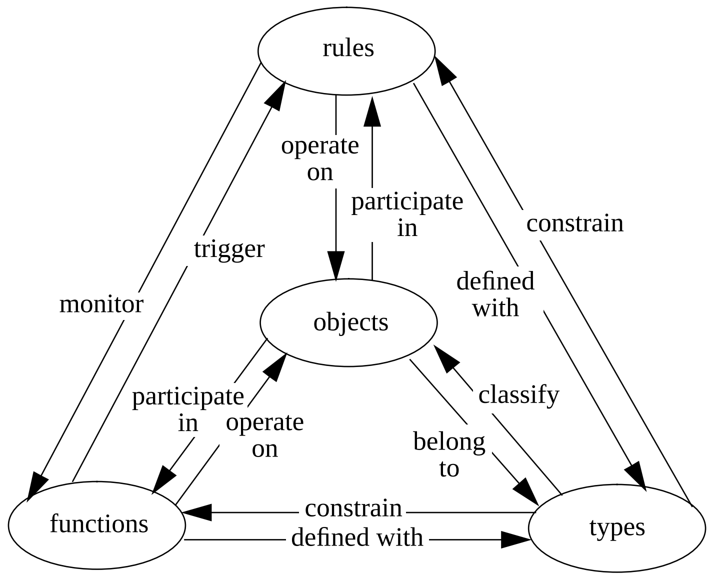


Figure 3.2: The AMOS data model

functions, e.g. =, < and >, and aggregate functions such as `sum` and `count`. Foreign functions written in some procedural language are currently also considered to be passive functions. Functions that are defined in terms of these functions can change, but never the passive functions themselves.[1]

The system currently does not have any purely active functions, but these would be event functions, i.e. functions that represent internal or external events. In some cases it is desirable to directly refer to specific events such as added or removed, this can be modelled as higher order event functions that change if tuples are added to their functional argument. Event functions that represent external changes are active foreign functions and can be sensor functions and time.

The rules presented here have conditions over stored and derived functions only. The events that triggers these conditions are the function update events, adding or removing tuples to/from functions. These functions can be seen as having both passive and active behaviour depending on whether they are referenced outside or inside rule conditions. Only functions without side-effects, i.e. queries, are allowed in rule conditions.

The rule processor calculates all the events that can affect a rule condition. This is the default for rule condition specifications and can be seen as a *safe* way to avoid that users forget specifying relevant events, as can happen with traditional ECA-rules. By allowing users to add specific event information through active functions specific events that system have not deduced can be

1. It would be strange to trigger on `1+1 = 3`

used for triggering rules as well. By allowing users to remove events that the system have deduced through negation of active functions, any event specification that can be specified in traditional ECA-rules can be specified more safely in CA-rules. The user can only remove events that he/she is aware of and events that are part of OO encapsulation will still trigger the rules correctly since these are deduced by the system. The extension of AMOSQL with event specifications through active functions would include introducing event operators, such as those defined in [17], into AMOSQL. Introducing active functions and extending AMOSQL with event modelling capability is future work.

By modelling rules as objects it is possible to make queries over rules. Overloaded and generic rules are also allowed, i.e. rules that are parameterized and can be activated for different types.

In AMOSQL, OSQL is extended with rules having a syntax conforming to that of OSQL functions. AMOSQL supports rules of CA type where the Condition is an OSQL query, and the Action is any OSQL procedure statement, except `commit`. Data can be passed from the Condition to the Action of each rule by using shared query variables, i.e. set-oriented Action execution[72] is supported.

The syntax for rules is as follows:

create rule *rule-name parameter-specification* **as**
    when *for-each-clause* | *predicate-expression*
    do *procedure-expression*
where
for-each-clause ::=
    for each *variable-declaration-commalist* **where** *predicate-expression*

The *predicate-expression* can contain any boolean expression, including conjunction, disjunction and negation. Rules are activated and deactivated by:

activate *rule-name* ([*parameter-value-commalist*]) [**priority** 0|1|2|3|4|5]
deactivate *rule-name* ([*parameter-value-commalist*])

Rules can be activated/deactivated for different argument patterns. The semantics of a rule are as follows: If an event of the database changes the truth value for some instance of the Condition to *true*, the rule is marked as *triggered* for that instance. If something happens later in the transaction which causes the Condition to become false again, the rule is no longer triggered. This ensures that we only react to *logical events*. The truth value of a condition is here represented by *true* for a non-empty result of the query that represents the condition and *false* for an empty answer, see section 4.1.

In the current implementation a simple *conflict-resolution* method, based on priorities, is used to specify the order of action execution of rules that are simultaneously triggered.

Some examples of AMOSQL rules are given below.

A classical example for active databases is that of monitoring the quantity of items in an inventory. When the quantity of an item drops below a certain threshold new items are to be automatically ordered.

```
create type item;
create type supplier;
create function quantity(item) -> integer;
create function max_stock(item) -> integer;
create function min_stock(item) -> integer;
create function consume_frequency(item) -> integer;
create function supplies(supplier) -> item;
create function delivery_time(item, supplier)
            -> integer;
create function threshold(item i) -> integer as
     select consume_frequency(i) * delivery_time(i, s)
                         + min_stock(i)
     for each supplier s where supplies(s) = i;
create rule monitor_item(item i) as
     when quantity(i) < threshold(i)
     do order(i, max_stock(i) - quantity(i));[1]
```

This rule monitors the quantity of an item in stock and orders new items when the quantity drops below the threshold (fig. 3.3) which considers the time to get new items delivered (where `order` is some procedure that does the actual ordering).The consume-frequency defines how many instances of a specific item are consumed on an average per day.

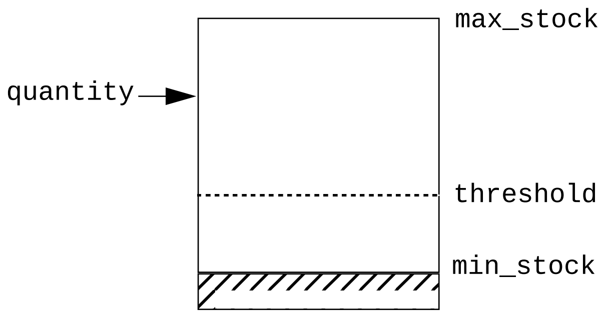


Figure 3.3: Monitoring items in an inventory

1. In AMOSQL `select` and `call` are syntactic sugar and are optional.

For example, the following definitions ensure that the quantity of shoelaces in the inventory is always kept between 100 and 10000 (if the supplier delivers on time) and will trigger the rule if the quantity drops below 140.

```
create item instances :shoelaces;
set max_stock(:shoelaces) = 10000;
set min_stock(:shoelaces) = 100;
set consume_frequency(:shoelaces) = 20;
create supplier instances :shoestring_inc;
set supplies(:shoestring_inc) = :shoelaces;
set delivery_time(:shoelaces, :shoestring_inc) = 2;
activate monitor_item(:shoelaces);
```

A rule that monitors all items can be defined as:

```
create rule monitor_all_items() as
     when for each item i
     where quantity(i) < threshold(i)
     do order(i, max_stock(i) - quantity(i));
```

In real life there will probably be several suppliers for one item. In that case the rules should really consider the minimum threshold, i.e. the supplier that can deliver fastest.

Another example of rules in active databases is that of *constraints*. If we want to ensure that the `quantity` of an item can never exceed the `max_stock` of that item, we can express that in the following rule.

```
create rule check_quantity() as
     when for each item i where
         quantity(i) > max_stock(i)
     do rollback;
```

The previous rules did not really use any of the OO capabilities of AMOSQL, i.e. there was only a flat set of user defined types. To illustrate these, take as an example a rule that ensures that no one at a specific department has a higher salary than his/her manager. Employees are defined to have a name, an income, and a department. The net income is defined based on 25% tax for both employees and managers, but with a bonus for managers of 100 before tax. Departments are defined to have a name and a manager. The manager of an employee is derived by finding the manager of the department to which the employee is associated. The rule `no_high` is defined to set the income of an employee to that of his/her manager if he/she has a net income greater than his/her manager. The AMOSQL schema is defined by:

```
create type department properties (name[1] charstring);
create type employee properties
      (name charstring, income number, dept department);
```

1. This is a short-hand for defining a stored function, `name`, on departments.

```
create type manager subtype of employee;
create function grossincome(employee e) -> number as
       select income(e);
create function grossincome(manager m) -> number as
       select income(m) + 100;
create function netincome(employee e) -> number as
       select employee.grossincome(e) * 0.75;
create function netincome(manager m) -> number as
       select grossincome(m) * 0.75;
create function mgr(department) -> manager;
create function mgr(employee e) -> manager as
       select mgr(dept(e));
create rule no_high(department d) as
    when for each employee e
    where dept(e) = d and
           employee.netincome(e) > netincome(mgr(e))
    do set employee.grossincome(e) = grossincome(mgr(e));
```

Note that the functions `grossincome`, `netincome`, and `mgr` are overloaded on the types `employee`, `manager`, and `department`, `employee`. For the function calls `grossincome(m)`, `grossincome(mgr(e))`, `netincome(mgr(e))`, `mgr(dept(e))`, and `mgr(e)` this is resolved at compile time, we call this *early binding*. This is possible since the actual parameters in the calls return distinct types. In cases when the compiler cannot deduce what function to choose, a dot notation, e.g. `employee.netincome(e)`, can be specified to aid the compiler to choose the correct function at compile time. In the rule condition, `employee.netincome` can be called for all employees, including managers, since managers are employees as well, but the condition will never be true for that case. This is because the `employee.netincome` would always be 100 less than `manager.netincome` for managers.

In cases when the compiler cannot deduce what function to choose, it will produce a query plan that does run-time type checking to choose the correct function, we call this *late binding*. This would be the case if `netincome` was not overloaded and `grossincome` was specified without dot notation. Different `grossincome` functions will then be chosen depending on if the argument it is called with is just an employee, or a manager as well. The rule condition would still be correct since if the `employee e` is a manager, the condition will never be true.

```
create function netincome(employee e) -> number as
       select grossincome(e) * 0.75;
create rule no_high(department d) as
    when for each employee e
    where dept(e) = d and
           netincome(e) > netincome(mgr(e))
    do set employee.grossincome(e) = grossincome(mgr(e));
```

This is because `manager.grossincome` would, in that case, be chosen in both instances in the condition and which then, obviously, would not be true. This rule is more elegant, but in order not to complicate the generated code and the discussion of change monitoring techniques in the following chapters, the first version of `no_high` will be used in the continuation of the example.

Also note that the `employee.grossincome` function is updatable since it is directly mapped to the stored function `employee.income`. The function `manager.grossincome` is not directly updatable since it cannot be directly mapped to a stored function. This is described in more detail in [51].

The `no_high` rule will be activated for a specific department and will serve as an example throughout the thesis.

Let us define a toys department with a manager and five employees:

```
create department(name) instances
       :toys_department("Toys")1;
create manager(name,dept,income) instances
       :boss("boss",:toys_department,10400);
set mgr(:toys_department) = :boss;
create employee(name,dept,income) instances
       :e1("employee1",:toys_department,10100),
       :e2("employee2",:toys_department,10200),
       :e3("employee3",:toys_department,10300),
       :e4("employee4",:toys_department,10400),
       :e5("employee5",:toys_department,10500);
```

The employees with their incomes and netincomes can be seen in fig. 3.4.

name	income	netincome
boss	10400	7875
employee1	10100	7575
employee2	10200	7650
employee3	10300	7725
employee4	10400	7800
employee5	10500	7875

Figure 3.4: Initial employee salaries

Now, if we activate the rule for the toys department and try to commit the trans-

1. This is a short-hand for setting the function `name`, for a department.

action a check is made if any of the employees have a netincome higher than their manager. No such employees exists and thus, the rule is not triggered.

```
activate no_high(:toys_department);
commit; /* check and commit */
```

Now if we change the income of employee2 and employee4:

```
set income(:e2) = 10600;
set income(:e4) = 10600;
```

Now we can see in fig. 3.5 that the netincomes of employee2 and employee4 exceeds that of their manager.

name	income	netincome
boss	10400	7875
employee1	10100	7575
employee2	10600	7950
employee3	10300	7725
employee4	10600	7950
employee5	10500	7875

Figure 3.5: Employee salaries before commit

If we try to commit this transaction the `no_high` rule will be triggered and the salaries of employee2 and employee4 will be set to that of their manager. This can be seen in fig. 3.6.

```
commit; /* check and commit */
```

name	income	netincome
boss	10400	7875
employee1	10100	7575
employee2	10500	7875
employee3	10300	7725

name	income	netincome
employee4	10500	7875
employee5	10500	7875

Figure 3.6: Employee salaries after commit

In this example, the rule condition monitoring consists of determining changes to the condition of the `no_high` rule. Changes to several stored functions (i.e. `dept`, `income,` and `mgr`) can affect the rule condition. In the example, only two updates are made to the `income` function. The rule condition monitoring must be efficient even if the number of employees is very large. However, evaluating the condition of `no_high` naively would result in checking the income of all employees for the department. Efficient techniques for evaluating rule conditions based changes that result from small updates, such as in these previous examples, will be discussed in the rest of the thesis.

3.3 Related Work

The data model of Iris is related to DAPLEX[66] and OODAPLEX[29]. DAPLEX is a functional data definition and manipulation language for database systems. DAPLEX introduced the concept of *derived functions* for defining *user views*. One difference is that in DAPLEX types are defined as functions as well. In OODAPLEX, DAPLEX is extended with objects that have identities independent of the values of their attributes and that *encapsulate* the operations of the object. Objects are grouped according to types, i.e. object class, and an inheritance mechanism is defined based on defining types in terms of supertypes.

The HiPAC[16][27] project introduced *ECA-rules* (Event-Condition-Action rules), where the Event specified when a rule should be triggered, the Condition was a query that was evaluated when the Event occurred, and the Action was executed when the Event occurred and the Condition was satisfied. In Ariel[41] the Event was made optional making it possible to specify *CA rules* which use only the Condition to specify *logical events* which trigger rules. Rules in OPS5[10] and monitors in [62] have similar semantics. In ECA rules the user has to specify all the relevant *physical events* in the Event part. Rules will not be triggered properly if the user forgets to specify some event. CA rules make physical events implicit, just as a query language makes database navigation implicit. Good evaluation and optimization techniques are required to make CA-rules as efficient as ECA-rules.

Our active rules [63] support the CA model by defining each rule as a pair, <Condition,Action>, where the Condition is a declarative AMOSQL query, and the Action is any AMOSQL database procedure statement. Data can be passed from the Condition to the Action of each rule by using shared query variables, i.e. set-oriented Action execution[72] is supported. Condition evaluation is normally delayed to a *check phase* usually at commit time. Immediate rule execution [27] is also possible, but is outside the scope of this thesis. In the check phase, change propagation is performed only when changes affecting activated rules have occurred, i.e. no overhead is placed on database operations (queries or updates) that do not affect any rules. After the change propagation, one triggered rule is chosen through a conflict resolution method. Then the action of the rule is executed for each instance for which the rule condition is true based on the Δ-set representing the changes of the rule condition.

The types of events that AMOSQL rules can be triggered on include internal events such as functional updates, creating/deleting objects, time related events, and external events (e.g. sensory updates). All event types will be included within the framework of CA-rules, however, this thesis discusses triggering on functional updates only. Work on a language for event specifications can be found in [17]. In our case this would be part of an extension of AMOSQL, instead of introducing a new language.

In [68] sensors are introduced as relations in a database system and as being *traced* or *sampled*. This is very much related to our view of passive and active functions. A traced sensor will be introduced as a *passive* function that is synchronously polled for changes. A sampled sensor will be introduced as an *active* function that displays asynchronous interrupt behaviour for signalling changes. Traced sensors can be used in queries and sampled sensors in rule conditions.

4 Condition Monitoring

4.1 Rule Semantics and Function Monitoring

The semantics of the rules in AMOS are based on function monitoring[62]. To be more specific, rules are based on the *when-function-changes-do-procedure* semantics(fig. 4.7).

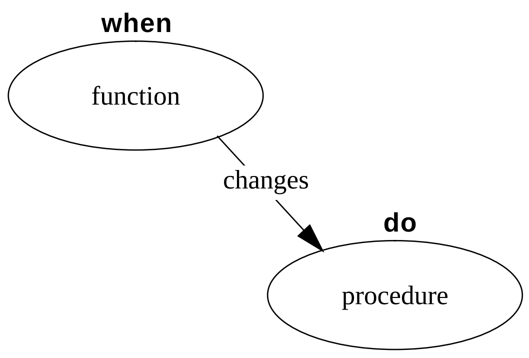


Figure 4.7: AMOSQL rule semantics

Take a rule r(x) defined as **when** c(x) **do** a(x).

This is a forward chaining rule that means 'execute a(x) when c(x) is evaluated to be true'. This is an imprecise definition of rule semantics, one really has to separate between *strict* and *nervous* rule semantics. Strict rule semantics for r would really be 'execute a(x) when c(x) is evaluated to be true after previously being false' and nervous rule semantics would be 'execute a(x) whenever c(x) is evaluated to be true regardless of whether it was true before'.

In order to explain how a rule is transformed into a function and a procedure, a new notation is introduced.

Forward chaining rules are written as:

<name>(<parameter-specification>) = (<condition> ⇒ <action>)

functions as:

<name>(<parameter-specification>) = select <return-specification>
where <predicate-expression>

and procedures as:

<name>(<parameter-specification>) = <procedure statements>

All parameters and heads of functions are subscripted with type information that specifies the types of the incoming parameters and the types of the returned values of functions, respectively.

We can now write the rule r as:

$r(x_{\text{type of x}}) = (c(x) \Rightarrow a(x))$,

where c(x) is a function call that returns a boolean value, i.e. $c(x_{\text{type of x}})_{\text{boolean}}$, and where a(x) is a procedure call. Note that x will be bound when the rule is activated.

By defining a *condition function* f that returns the type of x:

$f(x_{\text{type of x}})_{\text{type of x}}$= select x where c(x),

i.e. a function that returns a set of values of type x for all c(x) that return true, and an *action procedure* g that takes the type of x as argument,

$g(x_{\text{type of x}}) = a(x)$,

we can view rule condition monitoring as function monitoring of f, i.e. monitoring of changes to the set of values that f returns. Rule execution can then be defined for *nervous* rule behaviour as executing g on all the values of f, g(f(x)), and *strict* rule behaviour as executing g on the changes of f only, $g(\Delta f(x))$.

The condition of a rule can contain any logical expression and the action any logical expressions as well as side effects. For a rule

$r(x_{\text{type of x}}) = (c1(x) \,\&\, c2(y) \Rightarrow a1(x) \,\&\, a2(y))$,

and where c1(x) and c2(x) are boolean functions, i.e. $c1(x_{\text{type of x}})_{\text{boolean}}$, $c2(y_{\text{type of y}})_{\text{boolean}}$. The condition function to monitor is defined as:

$f(x_{\text{type of x}})_{\text{<type of x, type of y>}}$= select x, y where c1(x) & c2(y),

and the action procedure to execute is defined as:

$g(x_{\text{type of x}}, y_{\text{type of y}}) = a1(x) \,\&\, a2(y)$.

The semantics of rule execution is defined as g(f(x)) or $g(\Delta f(x))$. Note that x is here bound when the rule is activated, but y is free and fetched from the database.

Since functions are defined semantically as representing a set of values the rules are said to have set-oriented semantics, i.e. the rules monitor changes of a set that represents the condition and executes the action on the set that represents the changes to the condition set.

Some rules do not use the set-oriented semantics, as is the case with constraint rules that have actions that do transaction roll-backs. Such rules do not use any explicit values that have been produced in the condition when executing the action. Constraint rules are defined as:

$r(x_{\text{type of x}}) = (c(x) \Rightarrow \text{rollback}),$
$f(x_{\text{type of x}})_{\text{boolean}} = \text{select true where } c(x),$
$g(b_{\text{boolean}}) = \text{if b then rollback},$

The condition function f returns true if $c(x)$ returns a non-empty answer and false otherwise. The semantics of rule execution is defined as before, i.e. $g(f(x))$ or $g(\Delta f(x))$.

Since rules are objects of the type ‘rule’, the rule activation can be defined as a procedure

$\text{activate}(r_{\text{rule}}, l_{\text{list of object}})$

where r is a rule object and l is a list of objects that r is parameterized by. In the actual implementation the activate procedure is really defined as

$\text{activate}(r_{\text{rule}}, l_{\text{list of object}}, p_{\text{integer}})$

where p is the priority of the rule activation. Rule deactivation is defined likewise.

4.2 ObjectLog

AMOSQL functions are compiled into an intermediate language called ObjectLog[51]. ObjectLog is inspired by Datalog[14][71] and $\mathcal{LDL}$[21] but provides new facilities for effective processing of OO queries. ObjectLog provides a type hierarchy, late binding, update semantics, and foreign predicates.

- Predicate arguments are *objects*, where each object belongs to one or more *types* organized in a type hierarchy that corresponds to the type hierarchy of AMOS.
- Object creation and deletion semantics maintain the referential integrity of the type hierarchy.
- Update semantics of predicates preserve the type integrity of arguments. The optimizer relies on this to avoid dynamic type checking in queries.
- Predicates can be overloaded on the types of their arguments.
- Predicates can be further overloaded on the binding patterns of their arguments, i.e. on which arguments are bound or free when the predicate is evaluated.
- Predicates can be not only facts and Horn clause rules, but also optimized calls to invertible *foreign predicates* implemented in a procedural language. In the current system foreign predicates can be written in C.

- Predicates themselves as well as types are objects, and there are second order predicates that produce or apply other predicates. 2nd order predicates are crucial for late binding and recursion.

The translation from AMOSQL to ObjectLog consists of several steps(fig. 4.8). The *Flattener* transforms AMOSQL `select` statements into a flattened `select` statement where nested functional calls have been removed by introducing intermediate variables. The *Type checker* annotates functions with their type signatures in the *type adornment phase*, and finds the actual functions for overloaded functions (in case of early binding), or adds dynamic type checks (in case of late binding) in the *overload resolution phase*. The ObjectLog generator transforms stored functions into facts and derived functions become Horn clause rules. The *ObjectLog generator* also translates foreign functions into foreign predicates. The *ObjectLog optimizer* finally optimizes the ObjectLog program using cost based optimization techniques. More about the translation steps and the optimization techniques can be found in [51].

The optimized ObjectLog programs are currently interpreted, but work is in progress on compiling them for more efficient execution.

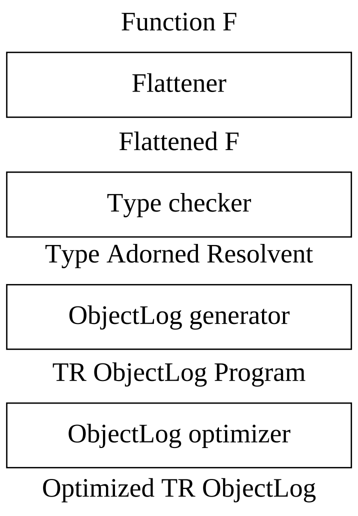


Figure 4.8: The translation of AMOSQL to ObjectLog

The transformations that are presented in section 5 can be done on either unoptimized or optimized ObjectLog programs. The resulting ObjectLog programs will need to be re-optimized in any case, see section 7.

4.3 Naive Change Monitoring

The condition in the first version of `no_high` rule is compiled into a condition function represented as an ordinary AMOSQL function, `cnd_no_high`, that returns all employees of a particular department with salaries higher than their manager:

```
create function cnd_no_high(department d) ->
                                    employee e as
       select e for each employee e
       where dept(e) = d and
               employee.netincome(e) > netincome(mgr(e));
```

Here, `netincome` is called with `mgr(e)` which means that the `manager.grossincome`, see section 3.2, (and consequently `income(m) + 100`) can be deduced in `netincome` at compile time since the function `mgr` always returns a manager. The query compiler transforms `cnd_no_high` to a derived relation (view) in ObjectLog[1]:

$$\begin{array}{l}
\texttt{cnd\_no\_high}_{\texttt{department,employee}}\texttt{(D, E)} \leftarrow \\
\qquad \texttt{mgr}_{\texttt{department,manager}}\texttt{(D, \_G1)} \wedge \\
\qquad \texttt{income}_{\texttt{employee,number}}\texttt{(\_G1, \_G2)} \wedge \\
\qquad \texttt{\_G3 = \_G2 + 100} \wedge \\
\qquad \texttt{\_G4 = \_G3 * 0.75} \wedge \\
\qquad \texttt{dept}_{\texttt{employee,department}}\texttt{(E, D)} \wedge \\
\qquad \texttt{income}_{\texttt{employee,number}}\texttt{(E, \_G5)} \wedge \\
\qquad \texttt{\_G6 = \_G5 * 0.75} \wedge \\
\qquad \texttt{>(\_G6, \_G4)}
\end{array}$$

Derived AMOSQL functions are compiled into derived relations and stored functions are compiled into stored relations (facts). When we hereafter use the term *relation* we use it interchangeably with the term *function*. The AMOSQL compiler expands as many derived relations as possible to have more degrees of freedom for optimizations. In the case of late binding full expansion is not always possible.

If the function `cnd_no_high` is evaluated with all the parameters to the rule instantiated, in this case with the $\texttt{D}_{\texttt{department}}$ instantiated, we can find the truth value for the condition and values of the free variables in the action. For the `no_high` rule, we get all the $\texttt{E}_{\texttt{employee}}$ for which the condition is true. The action part of the rule could then be executed for these truth values. The AMOSQL action procedure generated for the action in `no_high` looks like:

```
create function act_no_high(employee e) -> boolean as
       set employee.grossincome(e) = grossincome(mgr(e));
```

1. In ObjectLog Horn clauses are annotated with type names.

The execution of the action can be seen semantically as:

```
for each department d
where d = no_high_activations()
    call act_no_high(cnd_no_high(d));
```

If considering strict rule semantics we must find the changes to `cnd_no_high`:

```
for each department d
where d = no_high_activations()
    call act_no_high(Δcnd_no_high(d));
```

where `no_high_activations` is a function that returns all the arguments for which the `no_high` rule is activated.

4.4 Screener Predicates

If a transaction involves changes to functions that are referenced in a rule condition of some activated rule, it might be very expensive to evaluate the full condition every time in the check phase (usually at commit time). A better approach is to filter out changes that do not change the truth value of any activated rule condition. This can be done by generating *screener predicates* that are executed every time a specific function is updated, i.e. after the update is performed. If the update passes the screener predicate the change is saved and used in the check phase to determine what conditions to evaluate.

By generating screener predicates as queries, ordinary query optimization techniques can be used. How complex the predicate screeners should be depends on information such as the cost of evaluating the predicate and how often updates are performed, i.e. the update frequency of the base relation. A screener that is very restrictive, e.g. the complete rule condition, might be too expensive to execute every time a relation is updated while a screener that is too un-restrictive might cause unnecessary evaluation of rule conditions.

A *maximally discriminating* screener predicate for the `income` function can be defined as:

$$
\begin{array}{l}
\texttt{scr\_income}_{\texttt{employee}}\texttt{(E)} \leftarrow \\
\quad \texttt{no\_high\_activations}_{\texttt{department}}\texttt{(D)} \wedge \\
\quad \texttt{((mgr}_{\texttt{department,manager}}\texttt{(D, \_G1)} \wedge \\
\quad\quad \texttt{income}_{\texttt{employee,number}}\texttt{(\_G1, \_G2)} \wedge \\
\quad\quad \texttt{\_G3 = \_G2 + 100} \wedge \\
\quad\quad \texttt{\_G4 = \_G3 * 0.75} \wedge \\
\quad\quad \texttt{dept}_{\texttt{employee,department}}\texttt{(E, D)} \wedge \\
\quad\quad \texttt{income}_{\texttt{employee,number}}\texttt{(E, \_G5)} \wedge \\
\quad\quad \texttt{\_G6 = \_G4 * 0.75} \wedge \\
\quad\quad \texttt{>(\_G6, \_G4))} \vee \\
\quad\ \texttt{(mgr}_{\texttt{department,manager}}\texttt{(D, E)} \wedge \\
\quad\quad \texttt{income}_{\texttt{employee,number}}\texttt{(E, \_G7)} \wedge
\end{array}
$$

```
         _G8 = _G7 + 100 ∧
         _G9 = _G8 * 0.75 ∧
         dept_employee,department(_G10, D) ∧
         income_employee,number(_G10, _G11) ∧
         _G12 = _G11 * 0.75 ∧
         >(_G12, _G9)))
```

The no_high_activations is a function that returns all the departments for which the no_high rule is activated. This screener predicate checks if a particular update involves an employee at a department that the rule is activated for and if he/she gets a higher income than his/her manager, or if the update of the income of an employee involves a manager for a department that the rule is activated for and if there exists an employee at the same department with a higher income.

A *minimally discriminating* predicate screener for the income function can be defined as:

```
scr_income_employee(E) ←
       no_high_activations_department(D) ∧
       true
```

Neither of the above predicate screeners are satisfactory in most cases, the first predicate is too expensive to execute every time an update to the income is done, assuming no_high is activated for any department, and the second predicate causes uninteresting updates to slip through, causing unnecessary evaluation in the check phase. The goal is to find a predicate screener that is a good compromise between these two. By using cost information on the involved sub-expressions a good screener that is not too expensive can be found. For the income function a good candidate is:

```
scr_income_employee(E) ←
       no_high_activations_department(D) ∧
       (dept_employee,department(E, D) ∨
        mgr_department,manager(D, E))
```

This screener predicate checks if the income is changed for an employee that is a member or a manager of a department for which the no_high rule is activated for. If several predicate screeners are added to one function they are ordered in a single disjunction.

4.5 Incremental Change Monitoring

By studying the expanded execution plans in the no_high example we can see what updates of stored relations (i.e. stored AMOSQL functions) might affect the condition of a rule. The example rule depends on changes of `dept` (adding/removing employees to/from a department), `income` (changing employee or manager salaries), and `mgr` (changing the manager of a department). This can be modelled as a dependency network (fig. 4.9), where all the dependencies of a relation are modelled as sub-nodes.

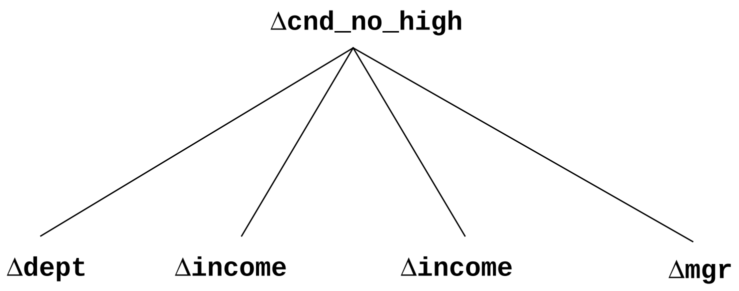


Figure 4.9: A dependency network for the rule `no_high`

A propagation network can then be generated from the dependency network (fig. 4.10) where the partial Δ-relations Δ`cnd_no_high`/Δ`dept`, Δ`cnd_no_high`/Δ`income`[1], and Δ`cnd_no_high`/Δ`mgr` denotes the influence of changes to the relations `dept`, `income` and `mgr` on the relation `cnd_no_high`.

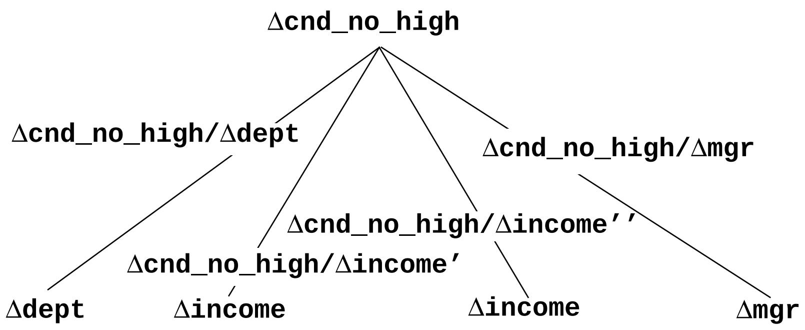


Figure 4.10: A propagation network for the rule `no_high`

A basic assumption is that most database transactions are short and the number of changes that affect activated rules is small. Therefore, evaluating the entire function that represents a rule condition is inefficient compared to evaluating a function that examines only the changes. In the example, the naive method

1. Two partial Δ-relations are created for `income`, one for each occurrence.

would check the salaries of all the employees in the given department. An incremental technique is preferable that checks, e.g., only employees that have had their income changed. By defining the above partial Δ-relations we can significantly reduce the cost of finding the changes to `cnd_no_high`. These partial Δ-relations are computationally equivalent to `cnd_no_high` with the exception that they each consider changes to one of its sub-relations instead of evaluating it in full. If several sub-relations changes then several partial Δ-relations will have to be evaluated and their results will have to be joined with a special join operator, $\cup_\Delta$.

In section 5 a calculus is presented based on the $\cup_\Delta$ operator for evaluating Δ-relations in terms of their partial Δ-relations. Here a separation is also made between positive and negative partial Δ-relations that checks additions and deletions separately.

The evaluation based on incremental change monitoring is by no means optimal in all situations. If for example there is a mix of updates that causes several partial Δ-relations to be evaluated it might be more efficient to evaluate the complete condition (as in the naive method). The choice of which method to choose in different situations is discussed in section 7.4.

4.6 Relating the Techniques

Screener predicates differ from Δ-relations in the way the are evaluated. Screener predicates are evaluated with top-down information flow since all the parameters to the relation are bound from the update information and any parameters from the rule activation are initially bound. Partial Δ-relations are evaluated with bottom-up information flow since not all of their parameters are initially bounded, the information from the Δ-set of one sub-relation is passed upwards to bind the parameters. The technique of generating screener predicates can be seen as pushing the condition downwards to the leaf nodes of the network. The screener predicates dynamically prune the network from uninteresting changes to avoid unnecessary propagation.

When generating screener predicates, the cost of the predicate screener must be compared with the sum of the cost of all the partial Δ-relations that must be evaluated because of the changes that slipped through the screener. To state it very simply one can define cost based screener predicates by a simple rule, ‘The higher total cost of the partial Δ-relations that are dependent on a particular Δ-set the higher complexity of the screener predicates can be motivated’. For a flat, i.e. bushy, propagation network as in the previous example, screener predicates are not really needed. They are more useful in deeper networks that result, e.g., from late binding. More on cost models for generating screener predicates can be found in section 7.2.

4.7 Related Work

In [12] a technique for detecting *Readily Ignorable Updates* (RIUs) to minimize execution of *alerters*, i.e CA-rules, is presented. *Construction diagrams* are constructed to determine how changes to base relations affect derived relations. Alerters are defined as *add-alerters* and *delete-alerters* that report additions or deletions of tuples, respectively. This is related to our techniques of using screener predicates to filter out ignorable updates and separating Δ-relations into positive and negative partial Δ-relations to check additions and deletions separately. Our techniques differ in that we use incremental evaluation techniques through change propagation and also that we use query optimization techniques when generating screener predicates and partial Δ-relations.

The technique of generating screener predicates is also related to that of *magic sets*[4] and *magic predicates*. However, magic predicates are used for limiting unnecessary evaluation during bottom-up query evaluation, while screener predicates are used for limiting propagation of changes. Screener predicates are generated using cost based information to limit their complexity. Screener predicates are not inserted into the body of a Horn clause, but are instead evaluated when a stored relation is updated. This can be compared to how *triggers* [3] are executed, except that screener predicates only determine if the change is interesting for any deferred rules, while triggers react immediately.

HiPAC[64] defined incremental propagation of Δ-relations through select-project-join. It is generalized in Sentinel [18]. HiPac used ECA rules, while our method uses CA rules with logical events where the physical events are calculated by the rule compiler. The method extinguishes complementary positive and negative physical events detected during a transaction. The *chain rule* of HiPAC[64] was defined as one large complete differential expression, while we are using a simpler partial differentiation when only a few functions are updated during a transaction. POSTGRES [69] and Starburst [53] both use ECA rules similar to HiPac. Starburst supports transition tables which correspond to Δ-relations or more precisely, Δ-sets, since they are defined only for stored relations, not for views.

Ariel [41] is implemented on top of POSTGRES and has CA-rules with similar semantics as our rules. Ariel uses a modified version of the OPS5 RETE algorithm called TREAT [55] for incrementally monitoring rule conditions. Tuples representing changes are propagated through the TREAT network. Tuples that satisfy some selection criteria in the network are stored in α-*memories*. Ariel avoids some materialization by using *virtual* α-*memories* that use derived relations instead of materialized ones. Instead of a technique that first does full materialization and then tries to avoid some materialization; this thesis presents a general calculus that avoids all unnecessary materialization. Strategic materializations are introduced through cost based optimization techniques. Our goal is furthermore a tightly coupled system that fully integrates rules into a query language, i.e. we want a propagation network that is integratable with our query execution mechanism, ObjectLog.

In [60] an extension of relational algebra with incremental relational expressions is presented. An outline of an algorithm for propagation of changes to the expressions is given. [60] defines two partial operations *disjoint union* and *contained difference* that assumes disjoint operands. In our approach this is maintained through the $\cup_\Delta$ operation on Δ-relations. By using Horn Clause Logic, i.e. Datalog[71] (or ObjectLog), to describe a calculus for incremental evaluation, it is more straight forward to implement since this is how query plans are represented in the system. The mapping of relational calculus to Datalog, or *domain calculus*, can be seen in the appendix. The update semantics of a system, e.g. set-oriented or bag-oriented semantics, affects in what order the propagation must be done. Introducing views in rule conditions introduces the need for cooperation in the propagation of changes affecting different conditions that depend on the same sub-expressions. Propagation of changes through a dependency network is not discussed at all in [60].

In [6] a method is presented that derives two optimized conditions from the original condition of a rule. The new conditions *Previously True* (PT) and *Previously False* (PF) are based on the knowledge of the previous truth value of the condition. Our rules are set-oriented and there is no single truth value for a rule condition, only a set of tuples for which the action of the rule is to be executed. An attribute grammar is also presented in [6] for implementing the approach, but nothing is said about how the actual propagation of changes is going to be done.

In the PARADISER system[30] incremental evaluation techniques are used for database rules processing. An algorithm for incremental evaluation of Datalog programs for the PARULEL rule language is presented. However, this is also a loosely coupled rule system built on top of a relational databases manager (POSTGRES and Sybase). The technique is based on *fact chains* that are stored and manipulated in the database and which are used to dynamically maintain the status of facts; no graph structure is constructed in memory. We believe that the rule system need to be tightly coupled with the database manager, using main memory data structures, for efficiency reasons. The cost model in a database manager must also be adopted to fit the new types of optimization issues introduced by rule processing.

5 A Formal Definition of Partial Differentiation

5.1 Incremental Evaluation

Incremental evaluation techniques was introduced in [59] as *Finite Differencing* of computable expressions. This is a technique that transforms programs into computationally equivalent programs that execute more efficiently by considering changes to expressions instead of executing them in full. A similar technique is used in [9] for efficiently maintaining materialized views. Efficient monitoring of rule conditions can be achieved by using similar techniques that considers changes to the conditions instead of performing full, naive evaluation.

Below follows a calculus for incremental evaluation of rule conditions. It formalizes the phases for update event detection and incremental change monitoring. The calculus is based on the usual set operators *union* ($\cup$), *intersection* ($\cap$), *difference* (-), and *complement* (~). Three new operators are introduced, *delta-plus* (Δ_+), *delta-minus* (Δ_-), and *delta-union* ($\cup_\Delta$). Δ_+ returns all that have been added to a set over a specified period of time, and Δ_- all that have been removed from the set. A *delta-set* (Δ-set) is defined as a tuple $<\Delta_+S, \Delta_-S>$ for some set S and $\cup_\Delta$ as the union of two Δ-sets. The operators are also used for discussing changes of bags that are more general than sets (set $\subset$ bag) since they allow duplicates.

The intuition behind the calculus of partial differentiation is presented in sections 5.2 and 5.3 with examples. In section 5.3.3 changes of *conjunction*, *disjunction*, and *negation* are formally defined in terms of set-operations. A justification for partial differentiation is given in the appendix, along with examples of partial differentiation of all the relational operators *union*, *difference*, *join*, *cartesian product*, *selection*, *projection*, *join*, and *intersection*.

Separate partial Δ-relations are generated for handling insertions and deletions. The intuition behind the calculus is to execute partial Δ-relations based on insertions of tuples in the state at the beginning of the propagation phase, since those tuples are present in the new state of the database. The partial Δ-relations based on deletions of tuple are executed in the state immediately after the previous propagation phase, since this represent the (old) state when the tuples were present in the database (fig. 5.1). The old state is calculated by performing a *logical rollback* that inverts all the updates of a specific relation.

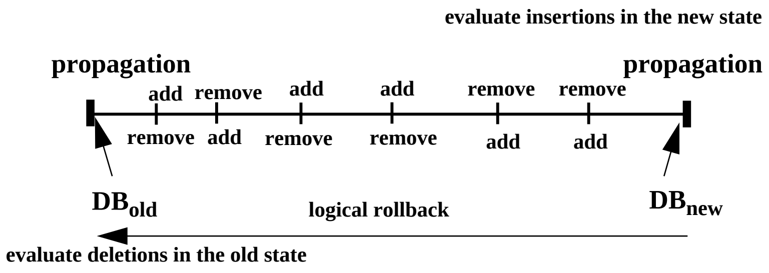


Figure 5.1: Evaluating positive v.s. negative changes

The calculus is based on accumulating all the relevant updates during a transaction. These accumulated changes are then used to calculate the partial Δ-relations which also involves calculating the old state of specific relations.

5.2 Update Event Detection

All changes to stored functions, i.e. base relations, in the database are logged in an undo/redo log. During database transactions, before physical update events are written to the log, a check is made if a stored base relation was updated that might change the truth value of some activated rule condition. If so, the *physical events* are inserted into a Δ-*set* that reflects all *logical events* of the updated relation. Since rules are only triggered by logical events the physical events have to be added with a special *delta union* operator, $\cup_\Delta$, that checks the contents of the Δ-set to see if each physical event has any effect or if it cancels out any old events in the Δ-set. We define the Δ-set of a base relation B by

$$\Delta B = <\Delta_+ B, \Delta_- B>,$$

where Δ_+B is the set of added tuples to B and Δ_-B is the set of removed tuples. We define $\cup_\Delta$ (i.e. $\Delta P \cup_\Delta \Delta Q$) informally by the table in fig. 5.2 where '+' is the addition of a tuple, '-' is the removal of a tuple, and '∅' means that the tuple is not in the Δ-set. The operator works correctly when there is no net effect of updates to a function.

If a change is made to an AMOSQL function value, the old value tuple is first removed and then the new is added.

ΔP \ ΔQ	∅	+	-
∅	∅	+	-
+	+	+	∅
-	-	∅	-

Figure 5.2: The $\cup_\Delta$ operator

For example, if we update the salary of some employee twice assuming that the income was originally 10100:

```
set income(:e1) = 10400;
set income(:e1) = 10100;
```

This corresponds to the physical update events:

```
-(income,:e1,10100),
+(income,:e1,10400),
-(income,:e1,10400),
+(income,:e1,10100).
```

The Δ-set for income changes accordingly with:

```
Δincome = <{},{(:e1,10100)}>
Δincome = <{(:e1,10400)},{(:e1,10100)}>
Δincome = <{},{(:e1,10100)}>
Δincome = <{},{}>
```

i.e. there is no net effect of the updates.

For bag-semantics, i.e. allowing duplicates, the Δ-set (or Δ-bag) must keep a count of duplicates and $\cup_\Delta$ must increment/decrement the count when adding positive/negative tuples (if the count becomes 0 then the tuple is removed).

5.3 Partial Differentiation

For monitoring changes of a given derived relation (view) P we need to define a Δ-relation ΔP. For stored relations, the Δ-relation is defined by its materialized Δ-set as above. For derived relations the Δ-set is defined as a pair:

$\Delta P = <\Delta_+ P, \Delta_- P>$, where
$\Delta_+ P = P - P_{old}$ and
$\Delta_- P = P_{old} - P$, and where
$P_{old} = (P \cup \Delta_- P) - \Delta_+ P$ for any relation P^1

This is a circular definition which is useless for anything but as a theoretical base for the following definitions. We need to define how to calculate the Δ-set of a derived relation in terms of the Δ-sets of the relations it depends on. The changes of the Δ-relation are materialized in a new Δ-set. This is a temporary materialization done in the propagation algorithm and is discarded as the propagation proceeds upwards. Changes, i.e. Δ-sets, that are not referenced by any partial Δ-relations further up in the network are considered as not needed any more. This assumes that there are no loops in the network, which is not the case with recursive relations, see section 8.5.

For efficient monitoring of rule conditions, the rule compiler generates several *partial Δ-relations* that detect changes to a derived relation given a change to one of the relations it is derived from. The technique is based on the assumption that the number of updates in a transaction is usually small and therefore only small effects on rule conditions will occur. Thus, the changes will only affect some of the partially differentiated relations. The change monitoring is separated into monitoring of positive changes (adding) and negative changes (removing).

5.3.1 Monitoring Positive Changes

For a relation P defined as a Horn clause with a conjunctive body, let D_p be the set of all relations that P depends on. Then the positive partial Δ-relations $\Delta P/\Delta_+X$ are defined by the body of P where a single relation $X \in D_p$ has been substituted by its positive Δ-relation Δ_+X.

For example, if

```
        p(X, Z) ←
                q(X, Y) ∧
                r(Y, Z)
```

then

```
        Δp(X, Z)/Δ+q ←
                Δ+q(X, Y) ∧
                r(Y, Z)
```

and

```
        Δp(X, Z)/Δ+r ←
                q(X, Y) ∧
                Δ+r(Y, Z)
```

In the example above the relations Q and R are either stored and the contents of ΔQ and ΔR are found by update event detection or, in the case of derived Δ-relations, found by evaluating changes to other partial Δ-relations in the same manner.

Let DB_{old} consist of the stored relations (facts)

```
q(1, 1)
r(1, 2)
r(2, 3)
```

1. The current database always reflects the new state

from p defined above we can derive

```
p(1, 2)
```

A transaction performs the updates

```
assert q(1, 2)
assert r(1, 4)
```

DB_{new} is now

```
q(1, 1)
q(1, 2)
r(1, 4)
r(2, 3)
```

from p we can derive

```
p(1, 2)
p(1, 3)
p(1, 4)
```

The updates gives the Δ-sets,

```
Δq = <{(1,2)},{}>
Δr = <{(1,4)},{}>
```

Evaluating Δp(X, Z)/Δ_+q and joining with $\cup_\Delta$ gives
Δp = <{1,3},{}>, i.e no changes are detected

Evaluating Δp(X, Z)/Δ_+r and joining with $\cup_\Delta$ gives
Δp = <{(1,3),(1,4)},{}>

In the no_high rule, there are two partial Δ-relations defined for changes to the cnd_no_high relation with respect to the income relation:

$\Delta\text{cnd\_no\_high}_{\text{department,employee}}(D, E)/\Delta_+\text{income}' \leftarrow$
$\quad \text{mgr}_{\text{department,manager}}(D, \_G1) \wedge$
$\quad \Delta_+\text{income}_{\text{employee,number}}(\_G1, \_G2) \wedge$
$\quad \_G3 = \_G2 + 100 \wedge$
$\quad \_G4 = \_G3 * 0.75 \wedge$
$\quad \text{dept}_{\text{employee,department}}(E, D) \wedge$
$\quad \text{income}_{\text{employee,number}}(E, \_G5) \wedge$
$\quad \_G6 = \_G5 * 0.75 \wedge$
$\quad >(\_G6, \_G4)$

$\Delta\text{cnd\_no\_high}_{\text{department,employee}}(D, E)/\Delta_+\text{income}'' \leftarrow$
$\quad \text{mgr}_{\text{department,manager}}(D, \_G1) \wedge$
$\quad \text{income}_{\text{employee,number}}(\_G1, \_G2) \wedge$
$\quad \_G3 = \_G2 + 100 \wedge$
$\quad \_G4 = \_G3 * 0.75 \wedge$
$\quad \text{dept}_{\text{employee,department}}(E, D) \wedge$

Δ_+`income`$_{\texttt{employee,number}}$`(E, _G5)` $\wedge$
`_G6 = _G5 * 0.75` $\wedge$
`>(_G6, _G4)`

Both of these partial Δ-relations are evaluated when changes occur to the `income` relation. The first one checks if the income of a manager was changed and if any employees have an income higher than his/her new income. The second one checks if the changes involves an employee and if the income of the employee is above that of the manager of his/her department.

Let us consider the example of the toys department again. Initially we have the incomes in fig. 5.3.

name	income	netincome
boss	10400	7875
employee2	10200	7650
employee4	10400	7800

Figure 5.3: Before updates

When we do the updates:
```
set income(:e2) = 10600;
set income(:e4) = 10600;
```
the updates passes through any screener predicate of `income` and we generate the Δ-set,

```
Δincome =
<{(:e2,10600),(:e4,10600)},
{(:e2,10200),(:e4,10400)}>
```

Then we have the incomes in fig. 5.4.

name	income	netincome
boss	10400	7875
employee2	10600	7950
employee4	10600	7950

Figure 5.4: After updates

Evaluating `cnd_no_high`$_{\texttt{department,employee}}$`(D, E)/`$\Delta_+$`income'` gives nothing since neither of the employees are managers. Evaluating `cnd_no_high`$_{\texttt{department,employee}}$`(D, E)/`$\Delta_+$`income''` and joining with $\cup_\Delta$ gives the Δ-set:

```
Δcnd_no_high =
<{(:toys_department,:e2),(:toys_department,:e4)},
{}>
```

and by evaluating:

```
act_no_high(Δ+cnd_no_high(:toys_department))
```

we get the final incomes in fig. 5.5.

name	income	netincome
boss	10400	7875
employee2	10500	7875
employee4	10500	7875

Figure 5.5: After rule execution

Note that this is actually not how the rule execution is implemented since we here ignore conflict resolution. The contents of

```
Δ+cnd_no_high(:toys_department)
```

is really saved in an *action-set* that eventually is used for action execution, see section 8. The negative changes are not needed in this example rule since the rule condition only depends on positive changes. If, however, some other rule triggers on the same or a similar condition and does a compensation that can cause the condition of this rule to become false, then the negative changes will have to be considered as well, see section 5.3.2.

For set-oriented semantics this method of propagation of positive changes might give a Δ-set that is too large, i.e. contains positive changes that existed in the old state of the database. This can lead to *nervous* rule behaviour because the Δ-set might cause a rule to be triggered even though the rule condition was already true before the changes occurred.

Take the example database:

```
t(11, 1)
s(X) ← t(Y, X), Y > 10
Δs(X)/Δ+t ← Δ+t(X), X > 10
```

From `s` we can derive `s(1)`.

The update

```
assert t(12, 1)
```

gives Δ`t = <{(12,1)},{}>` and
evaluating Δ`s(X)/`Δ_+`t` and joining with $\cup_\Delta$
gives Δ`s = <{(1)},{}>`.

From the last example it is easy to see why a partial Δ-relation can produce a set of changes that is too large. With set-oriented semantics we would have to check if `s(1)` was present in DB_{old} in order to have *strict* rule semantics. Strict rule semantics means that the rules are only triggered if the rule conditions become true after previously having been false. In many cases nervous rule behaviour is acceptable and this check will not have to be performed. For example, in rules that enforce some constraint, such as `no_high`, the natural thing to do in the action is to abort or compensate which will cause the condition to become false again. The `monitor_item` rule is an example of a rule that needs *strict* semantics, since a nervous `monitor_item` would make multiple orders, which is unacceptable. To avoid nervous rules it is necessary to inspect the old state of the relation representing the condition and to filter out positive changes of tuples that were already present.

5.3.2 Monitoring Negative Changes

In most cases a rule condition depends only on positive changes, as for the `no_high` rule. However, for negation and aggregation operators, see section 5.4, negative changes will have to be propagated as well. For strict rule semantics, propagation of negative changes is also necessary for rules that affect each other's rule conditions. A rule that is executed can produce negative changes that causes the condition of an already triggered, but not executed rule, to become false. This rule activation is then considered not to be triggered any more. This is explained in more detail in section 8.5.

The two partial Δ-relations of the relation P with regard to the negative changes of Q and R are defined as:

$$\begin{aligned}\Delta p(X, Z)/\Delta_- q \leftarrow\ & \\ & \Delta_- q(X, Y) \wedge \\ & r_{old}(Y, Z)\end{aligned}$$

and

$$\begin{aligned}\Delta p(X, Z)/\Delta_- r \leftarrow\ & \\ & q_{old}(X, Y) \wedge \\ & \Delta_- r(Y, Z)\end{aligned}$$

where $R_{old} = (\Delta_- R \cup R) - \Delta_+ R$
and since $\Delta_+ R \cap \Delta_- R = \emptyset$, i.e. $\Delta_- R - \Delta_+ R = \Delta_- R$, we have
$R_{old} = \Delta_- R \cup (R - \Delta_+ R) = \Delta_- R \cup (R \cap \sim\Delta_+ R)$, where
~ denotes set complement, and which can be expressed logically by:

$r_{old}(X, Y) \leftarrow$
$\quad \Delta_{-}r(X, Y) \vee$
$\quad (r(X, Y) \wedge \neg(\Delta_{+}r(X, Y)))$

where Q_{old} is defined likewise.

Let DB_{old} consist of the stored relations (facts)

```
q(1, 1)
r(1, 2)
r(2, 3)
```

from `p` defined above we can now derive

```
p(1, 2)
```

A transaction performs the updates

```
assert q(1, 2)
assert r(1, 4)
retract r(1, 2)
retract r(2, 3)
```

DB_{new} is now

```
q(1, 1)
q(1, 2)
r(1, 4)
```

from `p` we can now derive

```
p(1, 4)
```

The updates gives the Δ-sets,
`Δq = <{(1,2)},{}>` and
`Δr = <{(1,4)},{(1,2),(2,3)}>`.

Evaluating $\Delta p(X, Z)/\Delta_{+}q$ and joining with $\cup_{\Delta}$ gives
`Δp = <{},{}>`, i.e no changes are detected.

Evaluating $\Delta p(X, Z)/\Delta_{+}r$ and joining with $\cup_{\Delta}$ gives
`Δp = <{(1,4)},{}>`.

Evaluating $\Delta p(X, Z)/\Delta_{-}r$ and joining with $\cup_{\Delta}$ gives
`Δp = <{(1,4)},{(1,2)}>`.

Note that if we did not use the old state of `q` in $\Delta p(X, Z)/\Delta_{-}r$ we would get `Δp = <{(1,4)},{(1,2),(1,3)}>` which is clearly wrong. By propagating breadth-first, bottom-up we can calculate the old value of the database by doing a *logical rollback*, using the formulas above.

Take the database from the last example in the previous section:

```
t(11, 1)
```

```
s(X) ← t(Y, X), Y > 10
Δs(X)/Δ+t ← Δ+t(X), X > 10
Δs(X)/Δ-t ← Δ-t(X), X > 10
```

From `s` we can derive `s(1)`.
Now, let us assume the following updates instead,

```
assert t(12, 1)
retract t(11, 1)
```

These give `Δt = <{(12,1)},{(11,1)}>` and
evaluating `Δs(X)/Δ+t` and joining with $\cup_\Delta$ gives
gives `Δs = <{(1)},{}>`
and evaluating `Δs(X)/Δ-t` and joining with $\cup_\Delta$ gives
`Δs = <{},{}>`, since positive and negative tuples are extinguished by $\cup_\Delta$.

For set-oriented semantics, negative partial Δ-relations might also produce a Δ-set that is too large, i.e. deletions of tuples that are still present in the new state of the database. If we, for example, have `t(11,1)` and `t(12,1)` and retract one of them, we get a negative change of `s(1)`, but `s(1)` can still be derived from the database. Unlike for positive changes, this is more serious since it might cause rules not to trigger on positive changes since these have been cancelled by incorrectly propagated negative changes. To avoid this we have to check if the tuple is still present in the new state of the database. If this is not done the rules might under-react, which is unacceptable.

5.3.3 Combining Changes

The changes calculated in partial Δ-relations have to be combined before they can be propagated further in the network.

There exists an isomorphism f, denoted $\cong_f$, between the boolean algebra of ObjectLog and set algebra[1]:

$$f: \langle O, \neg, \wedge, \vee \rangle \rightarrow \langle 2^{At(O)}, \sim, \cap, \cup \rangle,$$

where O is the domain of objects in the database, ¬ is negation based on the Closed World Assumption, ∧ is logical conjunction, ∨ is logical disjunction, $2^{At(O)}$ is the power set of atoms in O, ~ is set complement, ∩ is set intersection, and ∪ is set union. Using this we can define change monitoring of ObjectLog through set operations.

Let Δ_+S, *delta-plus* of S, be the set of additions (positive changes) to a set S and Δ_-S, *delta-minus* of S, the set of deletions (negative changes) from S. Let the Δ-*set* (*delta-set)* of S be a tuple of the positive and the negative changes of a set S:

$$\Delta S = \langle \Delta_+S, \Delta_-S \rangle$$

We formally define the *delta-union*, $\cup_\Delta$, over Δ-relations as:

$$\Delta P \cup_\Delta \Delta Q = \langle (\Delta_+P - \Delta_-Q) \cup (\Delta_+Q - \Delta_-P), (\Delta_-P - \Delta_+Q) \cup (\Delta_-Q - \Delta_+P) \rangle$$

To detect changes of derived relations we define conjunction, disjunction, and negation in terms of their Δ-relations as:

$$\begin{aligned}\Delta(Q \wedge R) \cong_f \Delta(Q \cap R) = &\\ &<(\Delta_+Q \cap R) \cup (Q \cap \Delta_+R), \{\}>\\ &\cup_\Delta\\ &<\{\}, (\Delta_-Q \cap R_{old}) \cup (Q_{old} \cap \Delta_-R>\end{aligned}$$

$$\begin{aligned}\Delta(Q \vee R) \cong_f \Delta(Q \cup R) = &\\ &<(\Delta_+Q - R_{old}) \cup (\Delta_+R - Q_{old}), \{\}>\\ &\cup_\Delta\\ &<\{\}, (\Delta_-Q - R) \cup (\Delta_-R - Q)>\end{aligned}$$

$$\Delta(\neg Q) \cong_f \Delta(\sim Q) = <\Delta_-Q, \Delta_+Q>$$

Note that when there are changes to more than one part of a conjunction the definition above might give a set of changes that is too large, i.e. it might contain duplicates. For set-oriented semantics of relations this is no problem since all duplicates will be removed by $\cup_\Delta$. For bag-oriented semantics this is a serious problem since we can only disregard tuples that were generated from overlaps in the execution. This problem can be solved by adding checks that remove the overlaps:

$$\begin{aligned}\Delta(Q \wedge R) \cong_f \Delta(Q \cap R) = &\\ &<(\Delta_+Q \cap R) \cup ((Q - \Delta_+Q) \cap \Delta_+R), \{\}>\\ &\cup_\Delta\\ &<\{\}, (\Delta_-Q \cap R_{old}) \cup ((Q_{old} - \Delta_-Q) \cap \Delta_-R)>\end{aligned}$$

Such a technique is presented in [47] and can also be used as a general technique for optimizing partial Δ-relations.

For bag-oriented semantics, $\cup$ and $\cup_\Delta$ are commutative so the order of accumulation can be arbitrary. For set-oriented semantics, $\cup_\Delta$ has to be performed in the same order as the changes occurred in the transaction, see section 6.3. For disjunctions a check is made that positive/negative changes are propagated only if the other part of the disjunction was/is false (this check is only done for set-oriented semantics).

Note that the above definitions require that the Δ-sets of both Q and R are fully propagated before the new Δ-set can be computed, i.e. a breadth-first bottom-up propagation is crucial.

Next we define the *partial Δ-relation*, $\Delta P/\Delta X$, that incrementally monitors changes to P through changes of a single sub-relation X. *Partial Differentiation* of a relation is defined as generating partial Δ-relations for all the sub-relations of the relation. The net changes of partial Δ-relations are accumulated (using $\cup_\Delta$) into a Δ-set that materializes the changes represented by the Δ-relation. From the partial Δ-relations a dependency (propagation) network is generated where each node is a Δ-relation (fig. 4.10).

Let D_p be the set of all relations that P depends on. Positive partial changes are combined by:

$\Delta_+P = \bigcup \Delta P/\Delta_+X, \forall X \in D_p$
and negative changes by
$\Delta_-P = \bigcup \Delta P/\Delta_-X, \forall X \in D_p$
The full Δ-relation is defined as:
$\Delta P = <\Delta_+P, \{\}> \cup_\Delta <\{\}, \Delta_-P>$

For the example relation P in sections 5.3.1-5.3.2 the positive changes are combined by

$\Delta_+p(X, Z) = \Delta p(X, Z)/\Delta_+q \cup \Delta p(X, Z)/\Delta_+r$
and the negative changes by
$\Delta_-p(X, Z) = \Delta p(X, Z)/\Delta_-q \cup \Delta p(X, Z)/\Delta_-r$
and finally $\Delta p(X, Z) = <\Delta_+p(X, Z), \{\}> \cup_\Delta <\{\}, \Delta_-p(X, Z)>$

The order in which the accumulation of changes is done is important since $\cup_\Delta$ is not commutative for set-oriented semantics, see section 6.3.

For set-oriented semantics the partial Δ-relations might produce changes that are not really there. To avoid this we have to check that positive changes were not present in the old state of the database and that negative changes are not present in the current state of the database:

$\Delta P = <\Delta_+P - P_{old}, \Delta_-P - P>$

If this is not done, we might over/under-trigger rules since the positive/negative changes might be incorrectly propagated. If a check is not made for positive changes we will get *nervous* rule behaviour. For negative changes the check will always have to be done since under-triggered rules are undesirable. The old state of the database can be calculated using propagated changes or by using materialization techniques. By including checks of P_{old} and P directly into the partial Δ-relations the calculations will usually be more efficient than full re-calculations of P_{old} and P. The reason is that many variables will already be bound and there are usually many possibilities for optimizations of common sub-expressions. Another option is to materialize P, but this is always a trade-off between time and space, see section 7.5.

5.4 Changes to Aggregate Data

Changes of aggregates such as `notany` (logical $\neg\exists$, or negation as failure), `some` ($\exists$), `count` and `sum` will have to be defined through changes of incremental versions of the aggregation functions [8][47]. These are defined by saving a boolean value (for `notany` and `some`), a count (for `count`), or a sum (for `sum`) for each aggregate data and by using the changes to perform logical operations, increment/decrement the count or add/subtract from the sum. For aggregate functions both positive and negative changes will have to be propagated. For example, `count` will increment/decrement the counter associated with each aggregate data for positive and negative changes, respectively. The aggregate function `notany` is really a special case since it can be directly be determined by substituting positive changes for negative and vice versa, as was defined in section 5.3.3. If some tuple is added to the database then `notany` for that tuple will become false and if some tuple is removed from the database then `notany` for that tuple will become true.

All aggregation functions are, however, not incrementally computable, see [25].

5.5 Related Work

Work by [43] and [47] are very much related. In [43] rules are written directly in Datalog by referencing positive and negative Δ-relations directly. This makes it possible to write ECA style rules, but where the Event can be changes to any derived relation. In our system, the rule compiler generates partial Δ-relations, from AMOSQL CA style rules, that do incremental change monitoring of one sub-relation at a time. However, we plan to introduce higher-order functions that can refer to specific events, i.e. changes to any sub-function in the condition of a rule. The generated partial Δ-relations would then be very similar to the rules directly stated in [43].

The technique for incremental maintenance of materialized views proposed in [47] is also related and uses a technique similar to partial differentiation for change monitoring of a view (derived relation) to update a materialized view with the detected changes. However, our technique aims at avoiding view materialization. In [47] bag-oriented semantics are assumed and a technique for avoiding overlaps when executing several partial Δ-relations is presented. This technique is necessary for bag-semantics, but can be seen as an optimization of partial Δ-relations for set-oriented semantics.

6 Database Transactions and Update Semantics

6.1 Transactional Rules

When integrating active rules into a DBMS with transactional capabilities it is important to specify the semantics of the rules in terms of transactional behaviour.

The events that are detected and the consequent changes saved in Δ-sets will discarded in case of a transaction rollback. The changes are accumulated with $\cup_{\Delta}$ to capture the logical events instead of the physical. The actual changes to the Δ-sets are logged.

The semantics of rule definition and activation must also be specified in relation to transactions. Rules that are defined within a transaction are not permanent until the transaction is committed, hence a rollback will cause the rule to be removed. Likewise, a rule activation will be deactivated in the case of a rollback of the transaction in which the activation took place.

At rule creation, a condition function, an action procedure and partial Δ-relations are created. These are ordinary functions and relations and are thus also transactional. At rule deletion, the condition function, the action procedure, and the partial Δ-relations are removed. At rule activation the rule is integrated into the propagation network by inserting nodes for all the Δ-relations that the rule condition depends on. At rule deactivation the nodes are removed, if they are not shared by other rules. All operations on the network, e.g. insertions and deletions, have to be transactional.

If a rule is activated in the middle of a transaction the condition is naively evaluated in order for the rule to catch up with changes that occurred prior to the activation. The result is saved in the action-set of the rule activation. If the rule is rolled back then the action-set is discarded.

6.2 Rules that Perform Transaction Management

Rules can be used to enforce constraints over data by defining rules that abort transactions or perform compensating updates when changes occur that violates some constraint, i.e. causes the rule condition to become true.

In the case of *optimistic concurrency control* [49] rules can also be used to logically determine if updates of one transaction has interfered with some other transaction executing in parallel.

Another use for rules in transaction management is that of synchronizing parallel activities in the database. This can be done by defining rules that have conditions that specify the specific synchronization points and interacts with the activities (applications) in the action parts of the rules.

6.3 Update Ordering

If relations are defined to have set-oriented semantics then the order of accumulation of changes has to be the same as the changes occurred in the transaction. The $\cup_\Delta$ operator is not commutative when using set-oriented semantics. This is a problem related to deferred rules and does not concern immediate rules, since they do not accumulate any changes before they have their conditions evaluated.

Take, for example, a sequence of changes:

```
+(income,:e1,10400),
+(income,:e1,10400),
-(income,:e1,10400).
```

Assuming that the tuple is not originally present in the database, the second positive change has no effect at all while the third negative change causes an empty final net change. If the order is changed to:

```
+(income,:e1,10400),
-(income,:e1,10400),
+(income,:e1,10400),
```

then the final net change is

```
+(income,:e1,10400).
```

By defining a sequence to chronologically order all updates in a transaction, the sequence number, or *time stamp*, of each change can be propagated along with the changes. Before the changes of several partial Δ-relations are added to the complete Δ-relation (by $\cup_\Delta$) the changes have to be ordered according to their time stamps. To support this we redefine the incremental change propagation to also propagate the time of updates.

We define the *timed* Δ*-set* of a base relation B by:

$$\Delta B^T = <\Delta_+ B^T, \Delta_- B^T>$$

where $\Delta_+ B^T$ is the set of tuples and times when they were added to B and $\Delta_- B^T$ is the set of removed tuples and times when they were removed.

Take the example relation

```
            p(X, Z) ←
                        q(X, Y) ∧
                        r(Y, Z)
```

then

$$\Delta p^T(X, Z, T)/\Delta_+ q \leftarrow \Delta_+ q^T(X, Y, T) \wedge r(Y, Z)$$

where T is the time when a tuple was added and

$$\Delta p^T(X, Z, T)/\Delta_+ r \leftarrow q(X, Y) \wedge \Delta_+ r^T(Y, Z, T)$$

and

$$\Delta p^T(X, Z, T)/\Delta_- q \leftarrow \Delta_- q^T(X, Y, T) \wedge r_{old}(Y, Z)$$

where T is the time when a tuple was removed and

$$\Delta p^T(X, Z, T)/\Delta_- r \leftarrow q_{old}(X, Y) \wedge \Delta_- r^T(Y, Z, T)$$

We now define $\cup^T{}_\Delta$ to accumulate all changes and sort them according to their times before inserting them into a timed Δ-set. If two identical tuples are inserted into a Δ-set, the larger time stamp of the two is chosen.

Bags are more general than sets, i.e. set ⊂ bag, and have no restrictions against duplicate tuples. For bag-oriented semantics, the update order is unimportant since the Δ-bag has a count of all duplicate tuples. However, the propagation of the time of updates is also useful when introducing time events and temporal conditions into the active database, but this is outside the scope of this thesis.

6.4 Related Work

In HiPac[16], *coupling modes* (see section 2.3) were defined that specify the coupling between the event, condition and action parts of rule execution and the transaction(s) where they execute.

Constraints represented as assertions [3] specifies relationships that must hold after each transaction. Any transaction that violates a constraint will be aborted or the violating data will be changed through compensating updates. This behaviour can be attained by specifying production rules for constraint maintenance [15].

Sagas [37] can be used to specify *work flow* by defining a sequence of transactions together with compensating transactions that are to be executed in case of a saga rollback.

In [28] rules were used for defining work flow by having rules with decoupled semantics initiate new transactions. The rule conditions specify the criteria for starting work as separate transactions. In case of rollback rules can also be used to specify compensating actions. A special technique called *pipelining* is also discussed that sequentially orders subtransactions that have been generated from decoupled rules and that have triggered in some specific order.

No directly related work on update ordering has been found. In *Temporal*

Databases [67] there is usually a distinction between *transaction time*, *valid time* and *user-defined time*. Transaction time is the time when the information was stored in the database. Valid time is the time when a specific relationship in the database is valid. User-defined time is temporal information added by the user that is not supported by the database. In [23] *event time* was defined as the time when a certain event occurred in the real world and transaction time as the time when it was recorded in the database.

By registering events as changes to functions and storing them in timed Δ-sets the event time is transformed into transaction time. Only those events that are considered as potential changes to activated rule conditions are saved and consequently stamped with transaction time. Using this definition, the time which we use for ordering updates chronologically is transaction time based on the event times of insert and delete events. The technique of propagating transaction time through timed Δ-sets can be used for any timed events if they are associated with functional changes. More discussions on time concepts in temporal databases can be found in [48].

7 Optimization

7.1 General Optimization Techniques

The ObjectLog optimizer in AMOS as described in [51] is based on Horn clause rule substitution and a cost model for subgoal reordering. Horn clause rule substitution means that the optimizer combines Horn clause rules into larger rules by expanding subgoals. All subgoals cannot be expanded, e.g. late bound calls and recursion. By expanding all possible subgoals the following steps in the optimization process will have more degrees of freedom for optimization.

For any Horn Clause rule or predicate P, the input tuple is the tuple corresponding to the variable(s) that are bound in P. For a given input tuple there are zero, or several output tuples, corresponding to unbound variable(s) in P. Subgoal reordering is based on a cost model that calculates two cost estimates for P:

1. The *execution cost* of P, C_P, defined as the number of visited tuples, given that all variables of the input tuple are bound.

2. The *fan out*, Fp, which is the estimated number of output tuples produced by P for a given input tuple.

For a conjunctive query consisting of subgoals $\{P_i\}$, $1 \le i \le n$, the total cost C is calculated by the formula:

$$C = \sum_{i=1}^{n}\left(C_{P_i}\prod_{j=1}^{i-1}F_{P_j}\right)$$

For disjunctive queries, i.e. in disjunctive normal form, each part of the disjunction is optimized separately.

A *rank* for each subquery in a query plan is calculated by using fan out and cost information and some optimization strategy. The rank is used to reorder the subgoals in a query plan. In the system three different optimization strategies are available. A heuristic method based on calculating the ranks through a simple formula [51] is currently the default method. A randomized method based on Simulated Annealing and Iterative Improvement [46] is available as an option, and which is the most effective of the three for optimizing large queries, i.e. large join queries. Exhaustive optimization is also available [65] which calculates the optimal plan, but can only be used for smaller queries.

The fan out is currently defined by the following default values:

- $F_P = 1$ if the input tuple has a unique index.
- $F_P = 2$ if it has a non-unique index.
- $F_P = 4$ otherwise

The defaults for C_P are:

- $C_P = F_P$ if the input tuple has an index.
- $C_P = 100$ if it is unindexed, since the system has to scan the entire table.

Foreign predicates have by default $F_P = 1$ and $C_P = 1$, assuming they are cheap to execute and return a single result tuple. The user can provide cost hints for each predicate, which override the default assumptions about C_P and F_P. For Horn Clause rules F_P is calculated by using F_P of the subgoals.

The reordering of subgoals of a relation P, i.e. a Horn Clause, is performed by the optimizer by using the given C_Q and F_Q of each subgoal Q of P, with the aim of minimizing C_P.

7.2 Optimization of Screener Predicates

The screener predicate s described in section 4.4 can be optimized using the techniques described above. By calculating the total cost of the partial Δ-relations that are affected by the update, CP, we have the cost of letting an update slip through a screener predicate. For an update of Q affecting the partial Δ-relations $\{\Delta P_i/\Delta Q\}$, $1 \le i \le n$, we have:

$$CP = \sum_{i=1}^{n} C_{\frac{\Delta P_i}{\Delta Q}}$$

The total cost of a partial Δ-relation is really a recursive calculation since one partial Δ-relation, $\Delta P/\Delta Q$, can in turn affect other partial Δ-relations, $\Delta R_i/\Delta P$.

$$C_{\frac{\Delta P}{\Delta Q}} = \sum_{i=1}^{n} C_{\frac{\Delta R_i}{\Delta P}}$$

This calculation is done bottom-up using the dependency network, and stops when the rule conditions are reached.

The cost of the screener predicate scr_Q, C_{scr_Q} and the fan out F_{scr_Q} can be used to determine the effectiveness of a screener predicate. By minimizing CP * F_{scr_Q}, while keeping C_{scr_Q} * uf_Q < CP, an optimal screener predicate can been found, where uf_Q is a constant that considers the update frequency of Q. The constant uf_Q could be defined by some estimation determined through experiments or it could be changed continuously by using statistics on the numbers of updates per transaction of particular base relations. How to calculate the size of uf_Q is outside the scope of this thesis.

7.3 Optimization of Partial Δ-relations

When optimizing a partial Δ-relation the optimizer should take into account that the Δ-relation for which it is differentiated for is much smaller than the original relation. The ObjectLog optimizer described in [51] is being extended with new cost metrics for Δ-relations. Since Δ-relations usually are very small they will often be moved early. Often the Δ-relation will be placed first, however, this is not always the case as [43] assumes.

Take the partial Δ-relations generated for the condition of the `no_high` rule for monitoring changes of the income relation. The first partial Δ-relation:

$$
\begin{aligned}
&\Delta\texttt{cnd\_no\_high}_{\texttt{department,employee}}\texttt{(D, E)/}\Delta_{+}\texttt{income}' \leftarrow \\
&\qquad \texttt{mgr}_{\texttt{department,manager}}\texttt{(D, \_G1)} \wedge \\
&\qquad \Delta_{+}\texttt{income}_{\texttt{employee,number}}\texttt{(\_G1, \_G2)} \wedge \\
&\qquad \texttt{\_G3 = \_G2 + 100} \wedge \\
&\qquad \texttt{\_G4 = \_G3 * 0.75} \wedge \\
&\qquad \texttt{dept}_{\texttt{employee,department}}\texttt{(E, D)} \wedge \\
&\qquad \texttt{income}_{\texttt{employee,number}}\texttt{(E, \_G5)} \wedge \\
&\qquad \texttt{\_G6 = \_G5 * 0.75} \wedge \\
&\qquad \texttt{>(\_G6, \_G4)}
\end{aligned}
$$

is already optimal since $\texttt{D}_{\texttt{department}}$ is bound when the partial Δ-relation is evaluated and `mgr` is indexed on the department. The second partial Δ-relation:

$$
\begin{aligned}
&\Delta\texttt{cnd\_no\_high}_{\texttt{department,employee}}\texttt{(D, E)/}\Delta_{+}\texttt{income}'' \leftarrow \\
&\qquad \texttt{mgr}_{\texttt{department,manager}}\texttt{(D, \_G1)} \wedge \\
&\qquad \texttt{income}_{\texttt{employee,number}}\texttt{(\_G1, \_G2)} \wedge \\
&\qquad \texttt{\_G3 = \_G2 + 100} \wedge
\end{aligned}
$$

$$
\begin{array}{l}
\quad \texttt{\_G4 = \_G3 * 0.75} \wedge \\
\quad \texttt{dept}_{\texttt{employee,department}}\texttt{(E, D)} \wedge \\
\quad \Delta_{+}\texttt{income}_{\texttt{employee,number}}\texttt{(E, \_G5)} \wedge \\
\quad \texttt{\_G6 = \_G5 * 0.75} \wedge \\
\quad \texttt{>(\_G6, \_G4)}
\end{array}
$$

is not optimal. The Δ-relation should here be placed before the `dept` subgoal since `dept` has an index on employee and not on the department:

$$
\begin{array}{l}
\Delta\texttt{cnd\_no\_high}_{\texttt{department,employee}}\texttt{(D, E)/}\Delta_{+}\texttt{income''} \leftarrow \\
\quad \texttt{mgr}_{\texttt{department,manager}}\texttt{(D, \_G1)} \wedge \\
\quad \texttt{income}_{\texttt{employee,number}}\texttt{(\_G1, \_G2)} \wedge \\
\quad \texttt{\_G3 = \_G2 + 100} \wedge \\
\quad \texttt{\_G4 = \_G3 * 0.75} \wedge \\
\quad \Delta_{+}\texttt{income}_{\texttt{employee,number}}\texttt{(E, \_G5)} \wedge \\
\quad \texttt{dept}_{\texttt{employee,department}}\texttt{(E, D)} \wedge \\
\quad \texttt{\_G6 = \_G5 * 0.75} \wedge \\
\quad \texttt{>(\_G6, \_G4)}
\end{array}
$$

By defining cost hints for partial Δ-relations that specify, e.g. $F_P = 2$ and $C_P = 1$ this can be achieved.

Since all Horn clause rules have their subgoals fully expanded if possible, many opportunities for common subexpressions between different rule conditions are lost. By not doing full expansion common subexpressions can be achieved. In the `no_high` rule example the `cnd_no_high` could be retained as it is and allowing sharing of the `netincome` between different rule conditions:

$$
\begin{array}{l}
\texttt{cnd\_no\_high}_{\texttt{department,employee}}\texttt{(D, E)} \leftarrow \\
\quad \texttt{mgr}_{\texttt{department,manager}}\texttt{(D, \_G1)} \wedge \\
\quad \texttt{netincome}_{\texttt{manager,number}}\texttt{(\_G1, \_G2)} \wedge \\
\quad \texttt{dept}_{\texttt{employee,department}}\texttt{(E, D)} \wedge \\
\quad \texttt{netincome}_{\texttt{employee,number}}\texttt{(E, \_G3)} \wedge \\
\quad \texttt{>(\_G3, \_G2)}
\end{array}
$$

This would lead to a shared nodes in the propagation network. This node sharing is not done in the current implementation, except for late bound subgoals, i.e. when the correct functions cannot be determined at compile-time due to lacking type information. If query plans are not expanded many possible opportunities for optimization might be lost. This is a trade-off which has to be studied in more depth. Changing the optimizer to avoid full expansion in favour of common subexpressions is outside the scope of this thesis.

7.4 Incremental versus Naive Change Monitoring

As shown in section 9 the incremental change monitoring outperforms the naive one in the case of small updates that have small effects on rule conditions. In the case of large updates the incremental change monitoring performs badly, some times very badly. One possibility here, is to detect these large updates and automatically use naive change monitoring in these cases. This can, for example, be done by having a threshold size of Δ-sets that will be propagated. If the size exceeds the threshold, then the affected rule conditions will be evaluated naively. By deactivating rules when large updates are to be done and reactivating them before the transaction is to be committed, naive evaluation of the rule conditions can be attained. This can be done by deactivating rules when any of Δ-sets that they depend on exceed their threshold sizes. However, the cost of deactivating and activating rules will have to be taken into account. Deactivation and activation involves contracting and expanding the propagation network.

7.5 Logical Rollback versus Materialization

The choice between making a logical rollback versus using materialization to find the old value of a relation can be supported by cost information. The calculated execution cost of a relation P, C_P, can be compared to the calculated fan out, F_P. For a small C_P and a large F_P, logical rollback is advantageous. For a large C_P and a small F_P, materialization is a better decision. Exactly for what actual values of C_P and F_P the different choices should be made, has to be determined through further experiments.

7.6 Related Work

Since rule conditions are defined as ordinary queries, techniques for query optimization are relevant [46][51][65].

In [43] simple ad hoc optimization of Δ-relations is proposed, while it is here recognized as a more general optimization problem. In [41] techniques for finding common subexpressions are utilized.

8 Change Propagation

8.1 The Check Phase

In the *check phase* all the activated rules are checked to see if they are triggered. This is usually done at transaction commit. During ongoing transactions updates are saved in Δ-sets that are maintained for all stored relations that are referenced in any activated rules, see fig. 8.1. In the check phase these Δ-sets are propagated to form *action-sets* that contain all positive changes to the functions that represent the rule conditions. The Δ-sets are cleared during propagation as soon they are no longer needed. The action-sets are maintained until the transaction is actually committed, or aborted. After a round of propagation all activated rules with non-empty action-sets are inserted into a *conflict-set*. Then one rule is chosen, by some conflict-resolution method, and the action of the rule is executed for all the tuples in the action-set. After the rule action has been executed the action-set is cleared. The executed action might have caused changes to new Δ-sets, so these have to be propagated once more. This continues until the conflict-set is empty.

The action-sets of triggered rules are continuously updated while the rules are in the conflict-set. If an action-set becomes empty the rule is removed from the conflict-set.

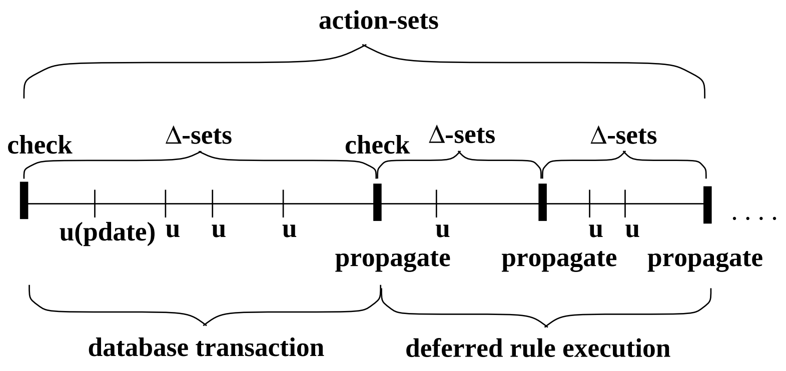


Figure 8.1: The phases of deferred rule execution

8.2 The Propagation Network

The propagation network contains all the information needed to propagate changes affecting activated rules. Since the propagation is done in a breadth-first, bottom-up manner the network can be modelled as a sequential list, starting with the lowest level and moving upwards. Each level consists of:

- A *change flag*, chg_flg, that marks a level as changed.
- A *list of network nodes*.

In fig. 8.2 the network, consisting of two levels, for the rule `no_high` can be seen.

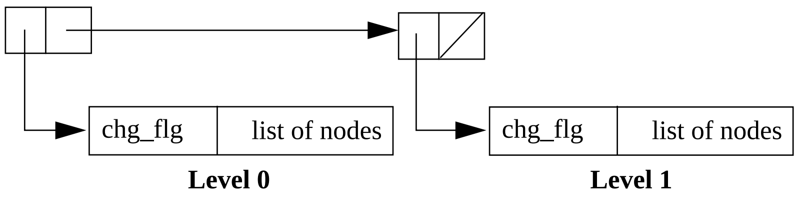


Figure 8.2: The propagation network for `no_high`

Each Δ-relation affecting activated rules is associated with one (and only one) node consisting of:

- A *change flag*, chg_flg, marking the node as changed.
- A *reference count*, cnt, that states how many nodes are dependent on this node.
- The Δ-*set* of the Δ-relation.
- A *list of affects nodes*, a-list, that are affected by changes to this node.
- A *list of depends on nodes*, d-list, together with the partial Δ-relations affected by the nodes below.
- A *pointer to the level* the node belongs to (not showed in fig. 8.3).
- A *list of pointers to rule activations* (if any) in the conflict set (not showed in fig. 8.3).

The number of levels needed in a network depends on how relations are expanded. For late binding extra levels will be inserted. The more levels in a network the more possibilities of node-sharing exist. This is discussed in section 7.3. How the nodes in the two levels are connected for `no_high` can be seen in fig. 8.3.

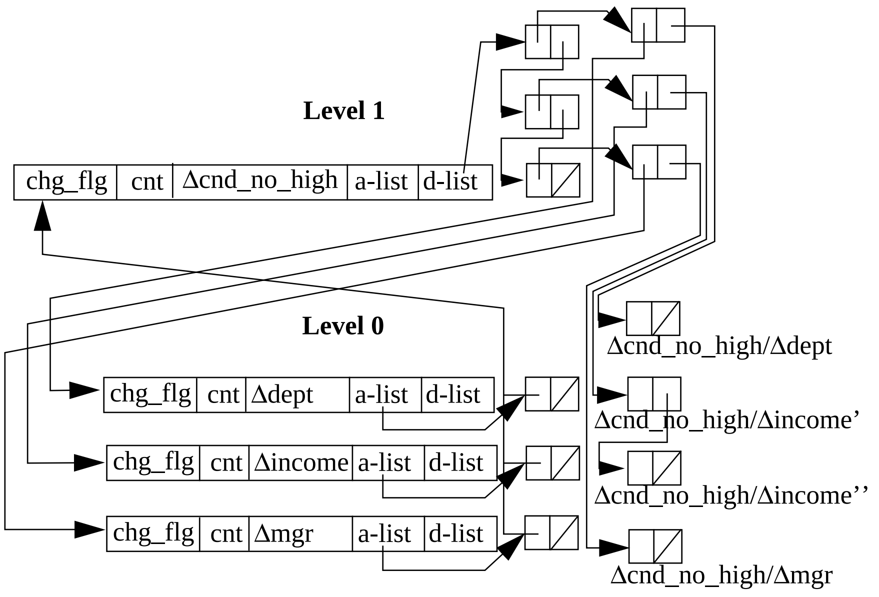


Figure 8.3: The nodes in the propagation network for `no_high`

8.3 Creation/Deletion of Rules

When a rule is created, a condition function and an action procedure is created (see section 4.1). When rules are created all partial Δ-relations of the rule condition are also generated (see section 5.3). Any re-optimization needed (see section 7.3) is also performed. When a rule is deleted, the condition function, the action procedure, and the partial Δ-relations are also deleted.

8.4 Activation/Deactivation of Rules

When rules are activated/deactivated the network is expanded/contracted with/without the nodes needed to propagate changes to the rule condition. When a rule is activated a naive evaluation of the rule condition for the specific activation pattern is performed. The result is saved in the action-set of the rule. This is done in order for the rule to catch up with all the changes that have occurred prior to the rule activation.

The algorithm for inserting Δ-relations into the network looks as follows:

```
Insert(ΔP):
    if ΔP is not already inserted into the network then
        create node_of(ΔP);
        if $D_P$ is empty, where $D_P$ is the set of relations that P depends on,
        then   /* P is a base relation */
               Insert_in_level(node_of(ΔP), 0);
        else
               for each ΔQ where $Q \in D_P$ do
                      Insert(ΔQ);
                      insert (node_of(ΔQ) . ΔP/ΔQ) into the
                      depends-on list node_of(ΔP).d-list;
                      insert node_of(ΔP) into the affects list
                      node_of(ΔQ).a-list;
               Insert_in_level(node_of(ΔP),
                   max(for each ΔQ where $Q \in D_P$: level_of(node_of(ΔQ))) + 1);

Insert_in_level(node, level):
    if level does not exist in network then create level;
    insert node into the level of the network;
    set level_of(node) = level;
```

The algorithm for removing Δ-relations looks as follows:

```
Remove(ΔP):
    if ΔP is present the network then
        if the affects list node_of(ΔP).a-list is empty then
            for each ΔQ where $Q \in D_P$
                remove (node_of(ΔQ) . ΔP/ΔQ) from the
                depends-on list node_of(ΔP).d-list;
                remove node_of(ΔP) from the affects list node_of(ΔQ).a-list;
                Remove(ΔQ);
            Remove_from_level(node_of(ΔP), level_of(node_of(ΔP)));
            delete node_of(ΔP);

Remove_from_level(node, level):
    remove node from level of network;
    if no nodes remain in the level then delete the level;
```

All operations to the network are transactional, i.e. the changes are logged so that they can be undone during a transaction rollback.

8.5 The Propagation Algorithm

During ongoing transactions all changes to the log are screened for changes that might affect activated rule conditions. If a change is made to a stored relation that has a corresponding node in level 0 in the propagation network, i.e. if a relevant update event is detected, then the change is added to the corresponding Δ-set (using $\cup_\Delta$).

In the check phase the propagation algorithm propagates all the non-empty Δ-sets in a breadth-first manner, as illustrated in fig. 8.4. Since the network is constructed in such a way that the change dependencies of one node, i.e. the Δ-relations it depends on, are calculated in the network levels below, a breadth-first propagation ensures that all the changes have been calculated when we reach the node.

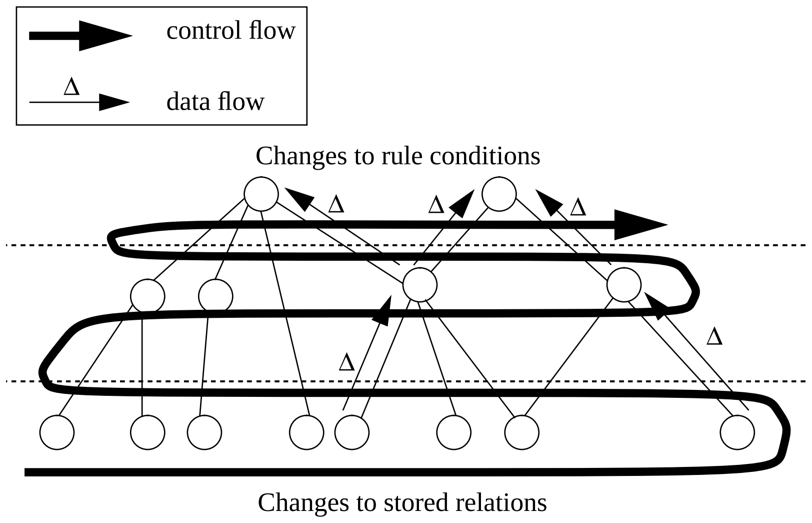


Figure 8.4: Propagation by a breadth-first algorithm

In the check phase one round of propagation is first done using the changes accumulated throughout the transaction. If any rules were triggered, i.e. were inserted into the conflict set in the propagation, then one rule activation is chosen, using some conflict resolution method. The action part of the chosen rule activation is then executed for each tuple generated in the condition of the rule. The action part is executed for each positive change since the last check phase, we call this the *action set*, which is calculated from the Δ-set of the condition. To determine if an already triggered rule (i.e. it is in the conflict set) is no longer triggered, the action set is saved and is modified continuously to determine if it is still triggered.

If a rule is triggered, a *rule activation* is inserted into the conflict-set. A rule activation consists of:

- The *rule* that was triggered
- The *action set* which contains the tuples on which the action is to be applied

The algorithm presented here is not dependent on any specific conflict resolution method. In the present implementation of rules in AMOSQL a simple priority scheme is used. To support this, each rule activation has a priority and the conflict set is divided into several priority levels. If the condition for which a rule was triggered changes to false, i.e. the action-set of a rule activation becomes empty, then the rule activation will be removed from the conflict set without executing the action.

Note that the algorithm presented here does not handle recursion, but can be extended to handle this. AMOSQL provides a *transitive closure*[1] operator that can handle most of the queries where recursive evaluation is needed. This operator is easier (or less difficult) to evaluate incrementally than general recursion since it involves looping over only one node in the network, see related work in section 8.6 and future work in section 10.

The log-screening looks as follows:

```
if a change is done to a stored relation with a corresponding node in the
propagation network then
    if the screener predicate of the relation evaluates to true then
            add the change to the $\Delta$-set of the node (using $\cup_\Delta$);
            set node.chg_flag = true;
            set (node.level).chg_flag = true; /* always level 0 */
```

The propagation algorithm looks as follows:

```
propagate():
    for each level in the network do /* starting with level 0 */
        if level.chg_flg then
            for each node in level.nodes do *
            if node.chg_flg then
                for each below-node in node.d-list do
                    if below-node.chg_flg then
                        execute each partial $\Delta$-relation and
                        accumulate the result into the $\Delta$-set
                        of the node (using $\cup_\Delta$); **
                        decrease_count(below-node);
                if node.a-list is empty then
                    /*  node is a top node */
                    for all activations of the rule do
                        calculate the action-set;
                        /* using the $\cup_\Delta$ and the $\Delta$-set */
                if node.$\Delta$-set has changed then
                    if node.$\Delta$-set is not empty and
                    node.a-list is not empty then
                        for each above-node in node.a-list do
                            set above-node.chg_flg = true;
                            set (above-node.level).chge_flg = true;
                            increase-count(node);
                    else /* node is a top node */
```

1. Transitive closure performs repetitive application of a function,
`tclose(`$f_{function}$`,`o_{object}`,`$n_{integer}$`) = `$f^n(o)$

```
                set node.chg_flg = false;
                clear node.Δ-set;
                for all activations of the rule do
                    if the action-set is not empty then
                        insert the rule activation into the
                        conflict-set; /* if it is not already there */
                    else /* the action-set is empty */
                        remove the rule activation from
                        the conflict-set; /* if it is there */
        level.chg_flag = false;

increase_count(node):
    set node.cnt = node.cnt + 1;

decrease_count(node)
    set node.cnt = node.cnt - 1;
    if node.cnt = 0 then
        set node.chg_flg = false;
        clear the node.Δ-set;
```

*) The algorithm can be modified to keep a separate list of all changed nodes in each level in order to avoid checking the change flag in all nodes. This will increase performance when the network becomes large.

**) If the system has set-oriented semantics the accumulated updates must be sorted in a chronological order.

The check phase looks as follows:

```
check():
    propagate();
    while conflict-set is not empty
        choose (using some conflict resolution method) and
        remove one rule activation from the conflict-set;
        execute the action on the changes of the rule condition
        (using the calculated action-set);
        clear the action-set of the executed rule activation;
        propagate();
```

8.6 Related Work

The PF-algorithm [43] is integrated with Datalog and uses incremental change monitoring of conditions by defining Δ-relations in a similar manner as in our approach. Unlike PF, we use a *breadth-first, bottom-up* propagation algorithm (as in [61]) to correctly and efficiently propagate both positive and negative changes without retaining space consuming materializations of intermediate Δ-relations between check phases. The PF-algorithm uses permanent materialization and propagates first negative changes then positive ones in a *depth-first* manner. Work on improving the PF-algorithm is presented in [40]. The algorithms presented in [43] and [47] can handle change propagation of recursive relations which our algorithm at the present can not.

Basic techniques on recursive query processing can be found in [5]. To handle recursive Δ-relations the propagation algorithm will have to be modified to return to previous levels in the network and to re-propagate the changes of the recursive Δ-relations using materialization and fixed-point techniques. However, since recursive queries are uncommon in AMOSQL (and in general [70]), the work has not been focused on recursion. When the network is constructed, loops can automatically be detected and a naive evaluation of the condition can be used instead.

In [34] an algorithm is presented that given as set of production rules, returns a set of the most profitable expressions that should be maintained. This work considers the effects of rule actions to other rule conditions. Our work, on the other hand, is concentrated on efficient monitoring of rule conditions. The incremental change monitoring technique can be seen as just an optimization since there is no semantic difference from naive evaluation. However, analysis of the rule actions as well could be an interesting extension to the technique.

Transitive closure [1] is a simplification of general recursion to mimic direct recursion where a function is continuously applied to its own result. Transitive closure can handle many of the queries that would otherwise be stated recursively in AMOSQL. Transitive closure is easier (less hard) to implement in the algorithm presented here since it involves looping over only one node in the propagation network, assuming the function is known at compile time. Techniques for efficient evaluation of transitive closure based on incremental evaluation techniques can be found in [25]. However, as mentioned in [25], incremental evaluation is not possible in all cases of transitive closures.

9 Performance

9.1 Performance Measurements of Change Monitoring

Performance of rule condition monitoring is related to expressability and the complexity of rule conditions. Expressability relates to the number of rules needed for a specific monitoring task. The rules in AMOSQL have the full expressability of AMOSQL queries in the condition. Complexity relates to the number of changes that can affect a rule condition and how they affect the condition. One rule can monitor several different changes to one rule condition. The incremental evaluation technique based on partial differentiation is efficient for small changes of a few functions that affect the rule condition, but is not so efficient for large changes to many such functions. The number of changes that can affect the rule condition does not directly relate to the efficiency of this technique since it only considers one change at a time. What affects performance is the number of changes and how a particular change affects the condition.

A series of measurements were made to determine how much more efficient or inefficient incremental evaluation is, based on partial differentiation and change propagation, compared to naive evaluation. Different kind of changes to a rule condition were also studied. The rule that was used for the measurements was the `monitor_all_items` rule in section 3.2:

```
create rule monitor_all_items() as
     when for each item i
     where quantity(i) < threshold(i)
     do order(i, max_stock(i) - quantity(i));
```

where the threshold function is defined as

```
create function threshold(item i) -> integer as
     select consume_frequency(i) * delivery_time(i, s)
                         + min_stock(i)
     for each supplier s where supplies(s) = i;
```

This rule monitors the changes in quantity of all items in an inventory. The rule condition depends on changes to the quantity of items, the consume-frequency of the items (how many items that are consumed on an average per day), the delivery time of items, the minimum stock of items and which supplier delivers a specific item. The first three measurements aimed at determining how much more efficient the incremental change monitoring is than the naive change monitoring for small changes. For the naive evaluation version, simple screener predicates were used. For the incremental evaluation version, the generated partial Δ-relations were optimized as described in section 7.3.

9.1.1 Benchmark 1

A series of 100 transactions were run where each transaction changed the quantity of one item. This will cause change to one partial Δ-relation which is very efficient to monitor by incremental techniques. The naive change monitoring technique will evaluate the whole condition regardless of how much have changed. The results can be seen in fig. 9.1. Note that the axis have logarithmic scale since the magnitude between the execution times of the different techniques is too great to display with linear scaled axis. The time for incremental change monitoring is very constant, regardless of the number of items, with an average time of 14 sec or 140 msec/transaction. Note that the times presented here do not represent the possible throughput of the AMOS architecture, only the results from running a prototype implementation as a regular application process.[1] The time for naive change monitoring increases linearly with the number of items and is on the average 8.2 sec/transaction for 10000 items.

From this first benchmark it is easy to see why incremental change monitoring is the better technique of the two, if the number of changes to a rule condition in a transaction is small. Naive change monitoring quickly becomes unfeasible as the size of the database grows and where rule conditions are complex queries over large portions of the database.

1. All measurements were made on a HP9000/710 with 64 Mbyte of main memory and running HP/UX.

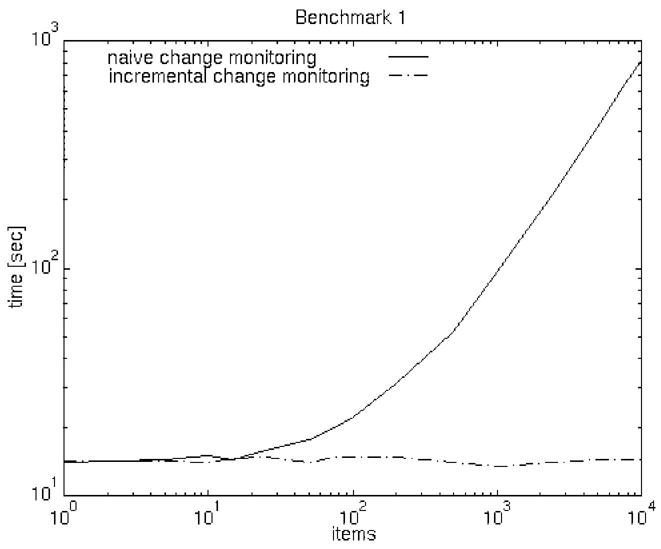


Figure 9.1: 100 transactions with 1 change to 1 partial Δ-relation

9.1.2 Benchmark 2

A series of 100 transactions were run where each transaction changed the quantity of one item and the delivery time for the item. This will cause change to two partial Δ-relations which is still very efficient to monitor by incremental techniques. The naive change monitoring technique will evaluate the whole condition and thus increases linearly with the size of the database. The results can be seen in fig. 9.2. The time for incremental change monitoring is on the average 15 sec or 150 msec/transaction.

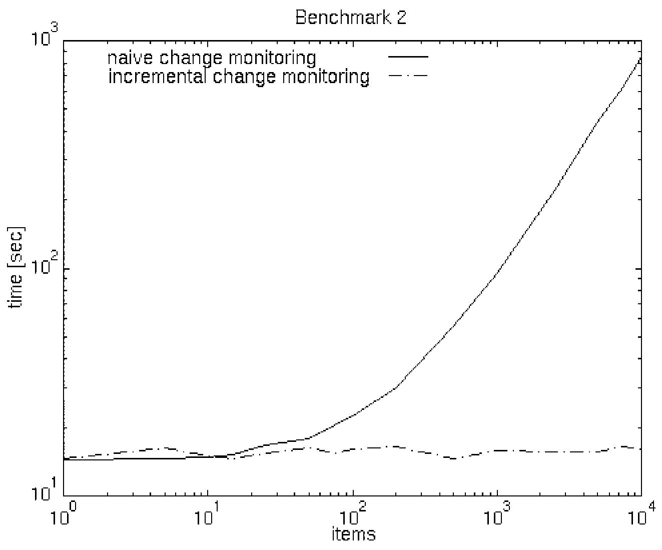


Figure 9.2: 100 transactions with 1 change to 2 partial Δ-relations

9.1.3 Benchmark 3

A series of 100 transactions were run where each transaction changed the quantity of one item, the delivery time for the item, and the consume-frequency for the item. This will cause change to three partial Δ-relations which is still very efficient to monitor by incremental techniques. The naive change monitoring technique will evaluate the whole condition and thus increases linearly with the size of the database. The results can be seen in fig. 9.3. The time for incremental change monitoring is on the average 16 sec or 160 msec/transaction.

These first measurements show that the incremental change monitoring technique is very efficient if the number of changes is small even if several parts of the rule condition are effected. As will be shown in benchmark 7 this is not always the case. It depends on how the change affects the condition and how expensive the related partial Δ-relation is to evaluate.

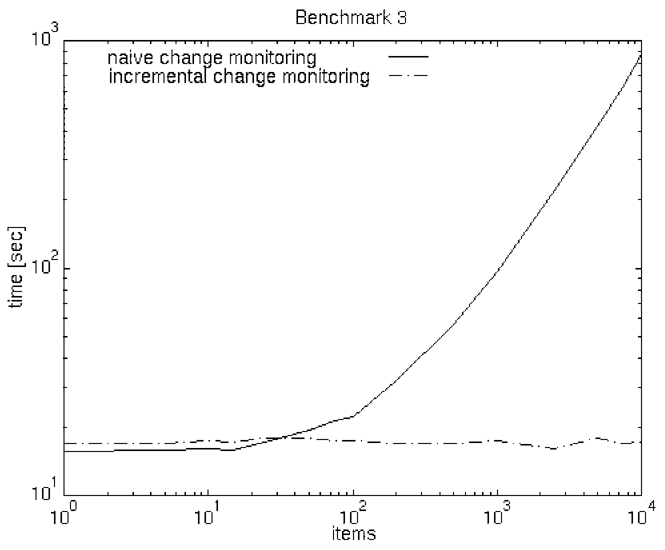


Figure 9.3: 100 transactions with 1 change to 3 partial Δ-relations

9.1.4 Benchmark 4

In this test one transaction was run which updated the quantity of all items in the database. This means that only one partial Δ-relation is affected. The affected partial Δ-relation has to check the quantities of all the items which is exactly what the naive change monitoring technique does. Since there is an overhead in doing the actual propagation, the incremental change monitoring technique performs slightly worse than the naive one (fig. 9.4).

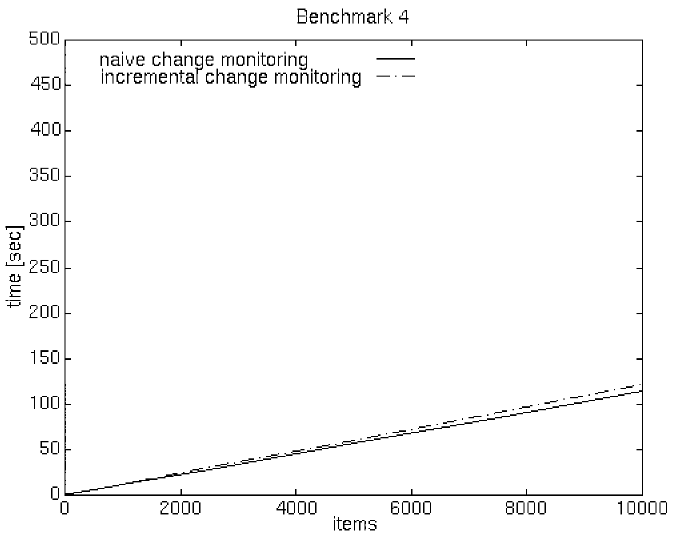


Figure 9.4: 1 transaction with n changes to 1 partial Δ-relation

9.1.5 Benchmark 5

In this test one transaction was run which updated the quantity and the delivery time of all items in the database. This means that two partial Δ-relations are affected. The affected partial Δ-relations have to check all the items which is exactly what the naive change monitoring technique does, but it does it all at once, as in the case of benchmark 4. The incremental change monitoring technique still performs only slightly worse than the naive one.

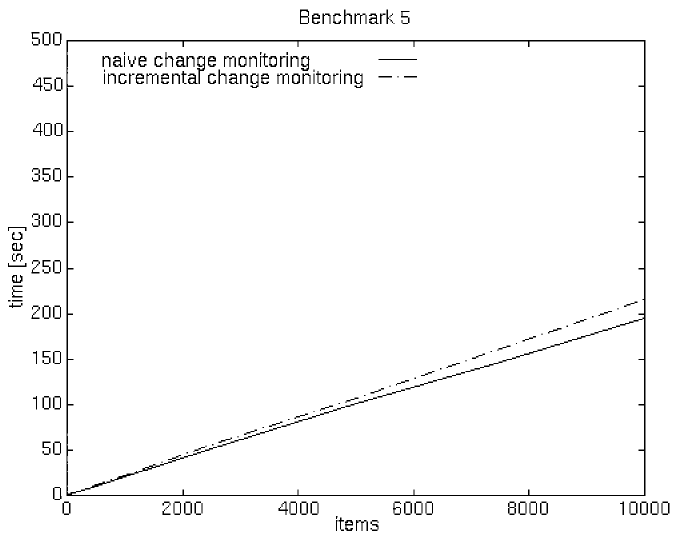


Figure 9.5: 1 transaction with n changes to 2 partial Δ-relations

9.1.6 Benchmark 6

In this test one transaction was run which updated the quantity, the delivery time, and the consume-frequency of all items in the database. This means that three partial Δ-relations are affected. The affected partial Δ-relations have to check all the items which is exactly what the naive change monitoring technique does, but it does it all at once, as in the case of benchmarks 4 and 5. The incremental change monitoring technique now performs much worse than the naive one. The reason for this can be found in the definition of the `threshold` function. Changing the consume-frequency causes a partial Δ-relation to be evaluated that has to check which supplier supplies the changed item, what is the delivery time of the item, and what is the minimum stock of the item. Changing the delivery time does not require finding the supplier since this is part of the propagated change.

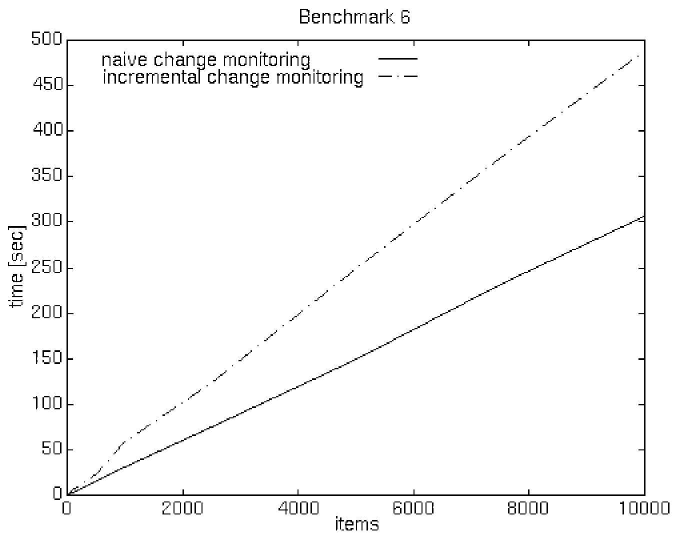


Figure 9.6: 1 transaction with n changes to 3 partial Δ-relations

9.1.7 Benchmark 7

The previous benchmark shows that different changes can have different effects on performance because the partial Δ-relations that they affect varies in cost of evaluation. The number of changes of a partial Δ-relation does not necessarily need to be large in order to have a large effect on the total performance. To highlight this we redefine the `min_stock` function to affect all items.

```
create function min_stock() -> integer;
create function threshold(item i) -> integer as
     select consume_frequency(i) * delivery_time(i, s)
                         + min_stock()
     for each supplier s where supplies(s) = i;
```

Changing the minimum stock does have a dramatic effect on performance. The quantity was changed for all items and the minimum stock was changed once in a single transaction fig. 9.7. This can be compared with fig. 9.4 to show that changing the minimum stock only once dramatically degrades performance for the incremental change monitoring technique.

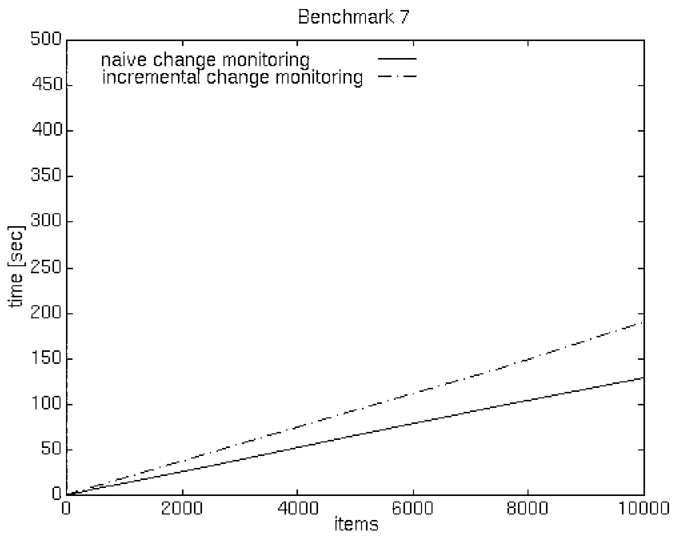


Figure 9.7: 1 transaction with n changes to 1 partial Δ-relation and 1 change to 1 expensive partial Δ-relation

From these measurements the conclusion can be made that the incremental change monitoring technique is superior for a small number of changes in most transactions. In this case, the performance is independent of the size of the database, we say that the incremental change monitoring *scale-up* with respect to size.

For a large number of changes in a transaction the naive change monitoring technique performs better. In the worst case, the incremental change monitoring, however, only performs worse than naive change monitoring by a constant factor. By deactivating rules with incremental change monitoring during large number of changes and activating them (causing a naive evaluation) before transactions are committed the best of both techniques can be attained. However, the cost of deactivating and activating a rule again must be considered here since this involves contracting and then expanding the propagation network.

Note that these measurements are not really dependent on that only one rule is activated. If several rules are activated, but only one of them is affected by the changes, i.e. if the condition refers to the function that changes, then there would be no overhead from the other rules. If, however, there are changes that affect several rules or if one rule causes changes that affects the condition of another rule then it is a different matter. Measuring performance in such cases requires carefully designed benchmarks that can give valuable information where the bottlenecks are in different change monitoring and action execution strategies.

9.2 Related Work

In [58] a performance test is presented for incremental updates in two different rule based programs. The first is the game of LIFE where incremental updates of a matrix of varying size is monitored. The second is a combinatorial optimization problem for allocating mortgage-backed securities. The results favours incremental update for the second program, but not for the first one. No real in-depth analysis is provided why this is so, only that updates in the first program produces major changes in the chain of inference which is unsuitable for incremental evaluation.

There is a need for development of a set of standard benchmarks that can be used for performance analysis of different evaluation strategies of rules.

10 Conclusions and Future Work

The thesis presents the *Object Relational* data model of AMOS and the introduction of active rules into AMOSQL, the query language of AMOS. The rules are based on the concept of function monitoring. All the changes in the system that the rules are to monitor will be introduced as changes to functions. Specific events that need to be referenced in rules will be introduced as higher order functions. The thesis presents work on rules that trigger on database updates only. Rules are of CA (Condition Action) type where the actual events that can trigger a rule are calculated by a rule compiler. The Condition of a rule can consist of an AMOSQL query and the action of AMOSQL procedure statements, i.e. queries and updates. Rules monitor changes to the rule conditions and data can be passed from the Condition to the Action of each rule by using shared query variables, i.e. set-oriented Action execution[72] is supported. By modelling rules as objects it is possible to make queries over rules. Overloaded and generic rules are also allowed, i.e. rules that are parameterized and can be activated for different types.

The thesis also presents techniques for efficient monitoring of changes to rule conditions. Rule condition monitoring must not decrease the overall performance to any great extent, with respect to either processor time or memory utilization. The following techniques for compilation and evaluation of rule conditions have been developed to meet these goals:

- To efficiently determine changes to all activated rule conditions, given updates of stored data, a *rule compiler* analyses rule conditions and generates change detection plans.
- To minimize unnecessary execution of the plans, *screener predicates* that screen out uninteresting changes are generated along with the change detection plans. The screener predicates are optimized using cost based query optimization techniques.
- For efficient monitoring of rule conditions, the rule compiler generates several *partially differentiated relations* that detect changes to a derived relation given changes to one of the relations it is derived from. The technique is based on the assumption that the number of updates in a transaction is usually small and therefore only small effects on rule conditions will occur. Thus, the changes will only affect some of the partially differentiated relations. The partially differentiated relations are optimized using cost based query optimization techniques.

- To efficiently compute the changes of a rule condition based on changes of sub-conditions, the partially differentiated relations are computed by *incremental evaluation* techniques [9] [59].
- To correctly and efficiently propagate both insertions and deletions (positive and negative changes) without unnecessary materialization or computation, the calculation of changes to a relation must be preceded by the calculation of the changes to all its sub-relations. This is accomplished by a *breadth-first, bottom-up* propagation algorithm, which also ensures graceful degradation as the complexity of rule conditions and as the size of the database increases.

An algorithm was presented as well as a performance study that compares the incremental evaluation technique with naive, full evaluation. The study showed that for small updates the performance is independent of the size of the database, we say that the change monitoring *scale-up* with respect to size.The main conclusion from the performance study can be summarized as: use incremental evaluation for small changes to rule conditions and use naive evaluation for large changes. By deactivating rules for large changes and activating them again at the end of transactions (causing a naive evaluation), the best of both techniques can be attained. When to automatically deactivate rules, or what handle to give the user for manual deactivation is open for further research.

There are several directions for possible future work on the rule system in AMOS. The types of events that the rules can trigger on needs to be extended to include schema updates, external events such as sensor updates, and time. This can be done by introducing active functions for all the changes that are desired to be monitored. Such functions already exist for querying the database schema. Extending AMOSQL with active functions for event specifications and event operations as in [17] is an important part of making AMOS a truly active database. Immediate rules are also needed, especially when introducing external asynchronous events and time events. How the incremental change monitoring techniques relate to time events must also be investigated further.

The incremental evaluation techniques presented here needs to be fully implemented to handle aggregates, transitive closure or recursion, and techniques for determine, or explicitly state, whether nervous rule semantics are sufficient for a particular rule or if strict semantics are needed.

The cost models for deciding between incremental versus naive change monitoring needs to be implemented and evaluated through empirical measurements. The same is needed for the cost models for choosing between doing a logical rollback or using materialization techniques for handling negative changes, i.e. database removals.

Further research is also needed in integration of AMOS with applications that utilize the rule system. Such work is important to give feedback on what functionality and extensions are needed in an active database system such as AMOS.

Appendix

Relational operations in Datalog

Datalog, or *domain calculus*, is equivalent to relational calculus in expressional power. The relational operations *union*, *difference*, *cartesian product*, *selection* and *projection* can be directly specified in Datalog. Other operations such as *join* and *intersection* that can be derived from these basic operations can also be directly specified.

Union

PARENT = FATHER $\cup$ MOTHER

is translated into

```
parent(X, Y) ← father(X, Y) ∨ mother(X, Y)
```

or

```
parent(X, Y) ← father(X, Y)
parent(X, Y) ← mother(X, Y)
```

Difference

FATHER = PARENT - MOTHER

is translated into

```
father(X, Y) ← parent(X, Y) ∧ ¬mother(X, Y)
```

Cartesian product

PAIR = PERSON $\times$ PERSON

is translated into

```
pair(X, Y) ← person(X) ∧ person(Y)
```

Selection

PAIR = $\sigma_{\$1 \neq \$2}(\text{PERSON}_{\$1} \times \text{PERSON}_{\$2})$

MILLIONAIRE = $\sigma_{\$2 > 999999}\text{INCOME}_{\$1,\$2}$

is translated into

```
pair(X, Y) ← person(X) ∧ person(Y) ∧ X ≠ Y
millionaire(X) ← income(X, Y) ∧ Y > 999999
```

Projection

IS_FATHER = $\pi_{\$1}\mathrm{FATHER}_{\$1}$

is translated into

```
is_father(X) ← father(X, Y)
```

Join

GRANDPARENT = $\pi_{X,Z}\mathrm{PARENT}_{X,Y} \bowtie \mathrm{PARENT}_{Y,Z}$ =

$\pi_{X,Z}(\sigma_{Y1=Y2}\mathrm{PARENT}_{X,Y1} \times \mathrm{PARENT}_{Y2,Z})$

is directly translated into

```
grandparent(X, Z) ←
            parent(X, Y1) ∧ parent(Y2, Z) ∧ Y1 = Y2
```

or more naturally expressed as

```
grandparent(X, Z) ← parent(X, Y) ∧ parent(Y, Z)
```

Intersection

RICH_GRANDPARENT = GRANDPARENT ∩ MILLIONAIRE =

(GRANDPARENT ∪ MILLIONAIRE) -

((GRANDPARENT – MILLIONAIRE) ∪

(MILLIONAIRE – GRANDPARENT))

is directly translated into

```
rich_grandparent(X) ←
            grandparent(X) ∨ millionaire(X) ∧
            ¬((grandparent(X) ∧ ¬millionaire(X)) ∨
              (millionaire(X) ∧ ¬grandparent(X))
```

or more naturally expressed as

```
rich_grandparent(X) ← grandparent(X) ∧ millionaire(X)
```

Justification for Partial Differentiation

Below follows a formal justification for the correctness of partial differentiation.

There exists an isomorphism f, denoted $\cong_f$, between the boolean algebra of ObjectLog and set algebra[1]:

$$f: \langle O, \neg, \wedge, \vee \rangle \rightarrow \langle 2^{At(O)}, \sim, \cap, \cup \rangle,$$

where O is the domain of objects in the database, $\neg$ is negation based on the Closed World Assumption, $\wedge$ is logical conjunction, $\vee$ is logical disjunction, $2^{At(O)}$ is the power set of atoms in O, ~ is set complement, $\cap$ is set intersection, and $\cup$ is set union. Using this we can define change monitoring of ObjectLog through set operations.

Let Δ_+S, delta-plus of S, be the set of additions (positive changes) to a set S and Δ_-S, delta-minus of S, the set of deletions (negative changes) from S. Let the Δ-*set* (delta-set) of S be a tuple of the positive and the negative changes of a set S:

$$\Delta S = \langle \Delta_+S, \Delta_-S \rangle$$

Let $\cup_\Delta$ (delta-union) be the operator that calculates the union of two Δ-sets:

$$\Delta P \cup_\Delta \Delta Q = \langle (\Delta_+P - \Delta_-Q) \cup (\Delta_+Q - \Delta_-P), \\ (\Delta_-P - \Delta_+Q) \cup (\Delta_-Q - \Delta_+P) \rangle$$

To detect changes of derived relations we define conjunction, disjunction, and negation in terms of their Δ-relations as:

$$\Delta(Q \wedge R) \cong_f \Delta(Q \cap R) = \\ \langle (\Delta_+Q \cap R) \cup (Q \cap \Delta_+R), \{\} \rangle \\ \cup_\Delta \\ \langle \{\}, (\Delta_-Q \cap R_{old}) \cup (Q_{old} \cap \Delta_-R \rangle$$

or for bag-oriented semantics

$$\Delta(Q \wedge R) \cong_f \Delta(Q \cap R) = \\ \langle (\Delta_+Q \cap R) \cup ((Q - \Delta_+Q) \cap \Delta_+R), \{\} \rangle \\ \cup_\Delta \\ \langle \{\}, (\Delta_-Q \cap R_{old}) \cup ((Q_{old} - \Delta_-Q) \cap \Delta_-R) \rangle$$

$$\Delta(Q \vee R) \cong_f \Delta(Q \cup R) = \\ \langle (\Delta_+Q - R_{old}) \cup (\Delta_+R - Q_{old}), \{\} \rangle \\ \cup_\Delta \\ \langle \{\}, (\Delta_-Q - R) \cup (\Delta_-R - Q) \rangle$$

$$\Delta(\neg Q) \cong_f \Delta(\sim Q) = \langle \Delta_-Q, \Delta_+Q \rangle$$

where $R_{old} = (\Delta_-R \cup R) - \Delta_+R$
and since $\Delta_+R \cap \Delta_-R = \varnothing$, i.e. $\Delta_-R - \Delta_+R = \Delta_-R$, we have
$R_{old} = \Delta_-R \cup (R - \Delta_+R) = \Delta_-R \cup (R \cap \sim\Delta_+R)$ which can be

expressed logically by:

$R_{old} = \Delta_{-}R \vee (R \wedge \neg(\Delta_{+}R))$

where Q_{old} is defined likewise.

Let D_p be the set of all relations that a relation P depends on. Let the *positive partial Δ-relations* $\Delta P/\Delta_{+}X$ of a relation P be defined by the body of P where a single relation $X \in D_p$ has been substituted by its positive Δ-relation $\Delta_{+}X$.

Let the *negative partial Δ-relations* $\Delta P/\Delta_{-}X$ of a relation P be defined by the body of P where a single relation $X \in D_p$ has been substituted by its negative Δ-relation $\Delta_{-}X$ and where all $Y \in D_p$, $Y \neq X$, have been substituted by Y_{old}.

Positive partial changes are combined by:

$\Delta_{+}P = \bigcup \Delta P/\Delta_{+}X, \forall X \in D_p$

and negative changes by

$\Delta_{-}P = \bigcup \Delta P/\Delta_{-}X, \forall X \in D_p$

The full *Δ-relation* (delta-relation) is defined as:

$\Delta P = \langle \Delta_{+}P, \{\} \rangle \cup_{\Delta} \langle \{\}, \Delta_{-}P \rangle$

Correctness is here defined as: given a relation P where D_p is the set of all other relations that P depends on and that we have all the net changes ΔS of all relations $S \in D_p$, then ΔP reflects the changes to P.

1. If P is a base relation then its changes can be found directly in ΔP.

2. If P is a derived, conjunctive relation then:
   i) If $P \leftarrow S \wedge T$ then we need to show that $\Delta P/\Delta_{+}S \leftarrow \Delta_{+}S \wedge T$ for all positive changes to S
   If T is a base relation then since the contribution of deduced facts in P are dependent on the facts both in S and T then any added facts in S that are also in T are also in P. In some cases and when using set-oriented semantics, added facts in S might give deduced facts that were already present in P_{old}, then the algorithm might cause *nervous* triggering of rules.To avoid this we have to calculate $\Delta P/\Delta_{+}S - P_{old}$. If T is a derived relation of n conjunctions then clearly:
   $\Delta P/\Delta_{+}S \leftarrow \Delta_{+}S \wedge T_1 \wedge ... \wedge T_n$
   If T is a derived relation of n+1 conjunctions then we also have:
   $\Delta P/\Delta_{+}S \leftarrow \Delta_{+}S \wedge T_1 \wedge ... \wedge T_{n+1}$
   and by induction the execution of positive, conjunctive partial Δ-relations has been shown to be correct.

   ii) If $P \leftarrow S \wedge T$ then we need to show that $\Delta P/\Delta_{-}S \leftarrow \Delta_{-}S \wedge T_{old}$ for all negative changes to S
   If T is a base relation then since the contribution of deduced facts in P are dependent on the facts both in S and T then any removed facts from S that also where in T_{old} supported facts are facts that are no longer in P. In some cases and when using set-oriented semantics, removed facts from S might

give deduced facts that are still present in P.To avoid incorrect propagation of negative changes we have to check that the deduced change is not still present in P, i.e. $\Delta P/\Delta_{-}S$ - P.
If T is a derived relation of n conjunctions then clearly:
$\Delta P/\Delta_{-}S \leftarrow \Delta_{-}S \wedge T_{old\ 1} \wedge ... \wedge T_{old\ n}$
If T is a derived relation of n+1 conjunctions then we also have:
$\Delta P/\Delta_{-}S \leftarrow \Delta_{-}S \wedge T_{old\ 1} \wedge ... \wedge T_{old\ n+1}$
and by induction the execution of negative, conjunctive partial Δ-relations has been shown to be correct.

3. If P is a derived, disjunctive relation, in disjunctive normal form, (and assuming set-oriented semantics), then:

i) If $P \leftarrow S \vee T$ then we need to show that $\Delta P/\Delta_{+}S \leftarrow \Delta_{+}S \wedge \neg T_{old}$ for all positive changes to S
If T is a base relation then since the contribution of deduced facts in P are dependent on facts in S or T then any added facts to S will cause positive changes to P if T was not already true for those facts.
If T is a derived relation of n disjuncts then clearly:
$\Delta P/\Delta_{+}S \leftarrow \Delta_{+}S \wedge \neg T_{old\ 1} \wedge ... \wedge \neg T_{old\ n}$
If T is a derived relation of n+1 disjuncts then we also have:
$\Delta P/\Delta_{+}S \leftarrow \Delta_{+}S \wedge \neg T_{old\ 1} \wedge ... \wedge \neg T_{old\ n+1}$
and by induction the execution of positive, disjunctive partial Δ-relations has been shown to be correct.

ii) If $P \leftarrow S \vee T$ then we need to show that $\Delta P/\Delta_{-}S \leftarrow \Delta_{-}S \wedge \neg T$ for all negative changes to S
If T is a base relation then since the contribution of deduced facts in P are dependent on facts in S or T then any removed facts from S will cause negative changes to P if T is not true for those facts.
If T is a derived relation of n disjuncts then clearly:
$\Delta P/\Delta_{-}S \leftarrow \Delta_{-}S \wedge \neg T_1 \wedge ... \wedge \neg T_n$
If T is a derived relation of n+1 disjuncts then we also have:
$\Delta P/\Delta_{-}S \leftarrow \Delta_{-}S \wedge \neg T_1 \wedge ... \wedge \neg T_{n+1}$
and by induction the execution of negative, disjunctive partial Δ-relations has been shown to be correct.

4. If P is a derived negated relation, $P \leftarrow \neg S$ then we need to show that:

i) $\Delta P/\Delta_{-}S \leftarrow \Delta_{+}S$
All facts not in S are deduced to be in P. If a fact is added to S then a negative change has to be deduced for P.

ii) $\Delta P/\Delta_{+}S \leftarrow \Delta_{-}S$
All facts in S are deduced to not be in P. If a fact is removed from S then a positive change has to be deduced for P.

5. If P is a derived relation that depends on the subrelations D_p then the changes

calculated by $\Delta P/\Delta_+X$ and $\Delta P/\Delta_-X$, $X \in D_p$, can be combined by $\cup_\Delta$ to give the total changes of P.

i) For set-oriented semantics $\cup_\Delta$ is defined as joining positive and negative changes in Δ-sets by removing duplicates and extinguishing complementary positive and negative changes.

ii) For bag-oriented semantics $\cup_\Delta$ is defined as joining positive and negative sets by keeping a count of duplicates and extinguishing complementary positive and negative changes. For conjunctions a modification of partial Δ-relations will also have to be done to remove overlaps in the execution [47]. Positive changes are the calculated by:
changing all subgoals y in $\Delta P/\Delta_+x$ to $y - \Delta_+y$, $\forall x, y \in D_p$ and $x \neq y$ and where y precedes x in the conjunction,
and negative changes by:
changing all y_{old} in $\Delta P/\Delta_-x$ to $y_{old} - \Delta_-y$, $\forall x, y \in D_p$ and $x \neq y$ and where y_{old} precedes x in the conjunction.

In the proof above an assumption was made that we have the net changes of the relation S collected in ΔS. The collection of changes of a relation was defined using the $\cup_\Delta$ operator. If relations are defined to have set-oriented semantics then the order of accumulation of changes has to be the same as the changes occurred in the transaction.

The proof above can be used for calculating incremental changes to the relational operators (with the related parts of the proof in parenthesis):

Union: (1, 3, 5)

$\texttt{parent(X, Y)} \leftarrow \texttt{father(X, Y)} \vee \texttt{mother(X, Y)}$

$\Delta\texttt{parent(X, Y)}/\Delta_+\texttt{father} \leftarrow \Delta_+\texttt{father(X, Y)} \wedge \neg\texttt{mother}_{old}\texttt{(X, Y)}$

$\Delta\texttt{parent(X, Y)}/\Delta_+\texttt{mother} \leftarrow \neg\texttt{father}_{old}\texttt{(X, Y)} \wedge \Delta_+\texttt{mother(X, Y)}$

$\Delta\texttt{parent(X, Y)}/\Delta_-\texttt{father} \leftarrow \Delta_-\texttt{father(X, Y)} \wedge \neg\texttt{mother(X, Y)}$

$\Delta\texttt{parent(X, Y)}/\Delta_-\texttt{mother} \leftarrow \neg\texttt{father(X, Y)} \wedge \Delta_-\texttt{mother(X, Y)}$

Difference: (1, 2, 4, 5)

$\texttt{father(X, Y)} \leftarrow \texttt{parent(X, Y)} \wedge \neg\texttt{mother(X, Y)}$

$\Delta\texttt{father(X, Y)}/\Delta_+\texttt{parent} \leftarrow \Delta_+\texttt{parent(X, Y)} \wedge \neg\texttt{mother(X, Y)}$

$\Delta\texttt{father(X, Y)}/\Delta_+\texttt{mother} \leftarrow \texttt{parent(X, Y)} \wedge \Delta_-\texttt{mother(X, Y)}$

$\Delta\texttt{father(X, Y)}/\Delta_-\texttt{parent} \leftarrow \Delta_-\texttt{parent(X, Y)} \wedge \neg\texttt{mother}_{old}\texttt{(X, Y)}$

$\Delta\texttt{father(X, Y)}/\Delta_-\texttt{mother} \leftarrow \texttt{parent}_{old}\texttt{(X, Y)} \wedge \Delta_+\texttt{mother(X, Y)}$

Cartesian product: (1, 2, 5)

$\texttt{pair(X, Y)} \leftarrow \texttt{person(X)} \wedge \texttt{person(Y)}$

$\Delta\texttt{pair(X, Y)}/\Delta_+\texttt{person'} \leftarrow \Delta_+\texttt{person(X)} \wedge \texttt{person(Y)}$

Δpair(X, Y)/Δ_+person'' $\leftarrow$ person(X) $\wedge$ Δ_+person(Y)

Δpair(X, Y)/Δ_-person' $\leftarrow$ Δ_-person(X) $\wedge$ person$_{old}$(Y)

Δpair(X, Y)/Δ_-person'' $\leftarrow$ person$_{old}$(X) $\wedge$ Δ_-person(Y)

Selection: (1, 2, 5)
millionaire(X) $\leftarrow$ income(X, Y) $\wedge$ Y $>$ 999999

Δmillionaire(X)/Δ_+income $\leftarrow$ Δ_+income(X, Y) $\wedge$ Y $>$ 999999

Δmillionaire(X)/Δ_-income $\leftarrow$ Δ_-income(X, Y) $\wedge$ Y $>$ 999999

Projection: (1, 5)
is_father(X) $\leftarrow$ father(X, Y)

Δis_father(X)/Δ_+father $\leftarrow$ Δ_+father(X, Y)

Δis_father(X)/Δ_-father $\leftarrow$ Δ_-father(X, Y)

Join: (1, 2, 5)
grandparent(X, Z) $\leftarrow$ parent(X, Y) $\wedge$ parent(Y, Z)

Δgrandparent(X, Z)/Δ_+parent' $\leftarrow$
 . Δ_+parent(X, Y) $\wedge$ parent(Y, Z)

Δgrandparent(X, Z)/Δ_+parent'' $\leftarrow$
            parent(X, Y) $\wedge$ Δ_+parent(Y, Z)

Δgrandparent(X, Z)/Δ_-parent' $\leftarrow$
            Δ_-parent(X, Y) $\wedge$ parent$_{old}$(Y, Z)

Δgrandparent(X, Z)/Δ_-parent'' $\leftarrow$
            parent$_{old}$(X, Y) $\wedge$ Δ_-parent(Y, Z)

Intersection: (1, 2, 5)
rich_grandparent(X) $\leftarrow$ grandparent(X) $\wedge$ millionaire(X)

Δrich_grandparent(X)/Δ_+grandparent $\leftarrow$
            Δ_+grandparent(X) $\wedge$ millionaire(X)

Δrich_grandparent(X)/Δ_+millionaire $\leftarrow$
            grandparent(X) $\wedge$ Δ_+millionaire(X)

Δrich_grandparent(X)/Δ_-grandparent $\leftarrow$
            Δ_-grandparent(X) $\wedge$ millionaire$_{old}$(X)

Δrich_grandparent(X)/Δ_-millionaire $\leftarrow$
            grandparent$_{old}$(X) $\wedge$ Δ_-millionaire(X)

References

[1] Abbott J. C.: Sets, Lattices and Boolean Algebra, *Allyn and Bacon*, 1969

[2] Aho A. V., Hopcroft J. E., Ullman J. D.: Data Structures and Algorithms, *Addison-Wesley*, 1983

[3] Astrahan M. M., et. al.: System R: Relational Approach to Database Management, *ACM Transactions on Database Systems*, Vol.1, No. 2, June 1976, pp. 97-137

[4] Bancilhon F., Maier D., Sagiv Y., Ullman J. D.: Magic sets and other strange ways to implement logic programs, *Proc. of the Fifth Symposium on Principles of Database Systems (PODS)*, Cambridge, MA, March 1986, pp. 1-15

[5] Bancilhon F., Ramakrishnan R.: An amateur's introduction to recursive query processing techniques, *Proc. of ACM SIGMOD 1986 Intl. Conf. on Management of Data*, Washington D.C, May 1986, pp. 16-52

[6] Baralis E., Widom J.: Using Delta Relations to Optimize Condition Evaluation in Active Databases, *Stanford rep. no. STAN-CS-93-1495*, Stanford University, 1993

[7] Beech D.: Collections of Objects in SQL3, *VLDB conf.* Dublin 1993, pp. 244-255

[8] Bernstein P.A., Blaustein B.T., and Clarke,E.M.: Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data, *VLDB 6*, Oct.1980, pp.126-136.

[9] Blakeley J.A., Larson P-Å., Tompa F.W.: Efficiently Updating Materializing Views, *ACM SIGMOD conf.*, Washington D.C., 1986, pp. 61-71.

[10] Brownston L., Farell R., Kant E., Martin A.: Programming Expert Systems in OPS5, *Addison-Wesley, Reading Mass.* 1986

[11] Buchman A. P., Branding H., Kudrass T., Zimmermann J.: REACH: a REal-time, ACtive and Heterogeneous mediator system, *IEEE Data Engineering bulletin*, Vol. 15, No. 1-4, Dec. 1992, pp. 44-47

[12] Buneman O.P., Clemons E. K.: Efficiently Monitoring Relational Databases, *ACM Trans. Database Syst.* 4.3 , July 1979, pp. 368-382

[13] Ceri S., Fraternali P., Paraboschi S., Letizia T.: Constraint enforcement through production rules: putting active databases at work, *IEEE Data Engineering bulletin*, Vol. 15, No. 1-4, Dec. 1992, pp. 10-14

[14] Ceri S., Gottlib G., Tanca L.: What You Always Wanted to Know About Datalog (And Never Dared to Ask), *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 1, March 1989

[15] Ceri S., Widom J.: Deriving Production Rules for Constraint Maintenance, *VLDB conf.*, Brisbane, Aug. 1990, pp. 566-577

[16] Chakravarthy S., et. al.: HiPAC: A Research Project in Active Time-Constrained Database Management, *Xerox Advanced Information Technology*, Cambridge, Mass., July 1989

[17] Chakravarthy S., Mishra D.: An Event Specification Language (Snoop) for Active Databases and its Detection, *UF-CIS Technical Report, TR-91-23*, Sept. 1991

[18] Chakravarthy S., Garg S.: Extended Relational Algebra (ERA) for optimizing situations Active Databases, *Tech. Report UF-CIS TR-91-24*, CIS Dept., Univ. of Florida, Gaines-

ville, Nov. 1991.

[19] Chakravarthy S., Navathe S. B., Garg S., Mishra D., Sharma A.: An Evaluation of Active DBMS Developments, *UF-CIS Technical Report* TR-90-23, University of Florida, 1990

[20] Chandra R., Segev A.: Active Databases for Financial Applications, *RIDE ´94*, Houston, Febr., 1994, pp. 46-52

[21] Chimenti D., Gamboa R., Krishnamurthy R.: Towards an Open Architecture for $\mathcal{LDL}$, *15th VLDB conf.*, 1989, pp. 195-205

[22] Clark J.J, Yuille M.I.: Data Fusion for Sensory Information Processing Systems, *Kluwer Acad. Publ.*

[23] Copeland G., Maier D.: Making Smalltalk a Database System, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Boston, MA, 1984

[24] Cornelio A., Navathe S.B.: Using Active Databases for Real Time Engineering Applications, *Proc. 9th Intl. Conf. on Data Engineering*, Vienna, April, 1993

[25] Dar S., Agrawal R., Jagadish H. V.: Optimization of Generalized Transitive Closure Queries, *Proc. 7th IEEE Intl. Conf. on Data Engineering,* Kobe, Japan, April, 1991, pp. 345-354

[26] Dayal U., Buchman A.P., McCarthy D.R.: Rules are objects too: A Knowledge Model for an Active, Object-Oriented Database System, *Proc. 2nd Intl. Workshop on Object-Oriented Database Systems*, Lecture Notes in Computer Science 334, Springer 1988

[27] Dayal U., McCarthy D., The architecture of an Active Database Management System, *ACM SIGMOD conf.*, 1989, pp. 215-224

[28] Dayal U., Hsu M., Ladin R.: Organizing Long-Running Activities with Triggers and Transactions, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Atlantic City, May 1990

[29] Dayal U., Queries and Views in an Object-Oriented Data Model, *Proc. of the 2nd Intl. Workshop on Database Programming Languages*, Oregon, 4-8 June 1989, pp. 80-102

[30] Dewan H. M., Ohsie D., Stolfo S. J., Wolfson O., Da Silva S.: Incremental Database Rule Processing in PARADISER, *Journal of Intelligent Information Systems*, 1:2, 1992

[31] Doyle J., Truth maintenance systems for problem solving, *MIT AI Laboratory Technical Report 419*, Cambridge, 1978

[32] Erman L. D., Hayes-Roth F., Lesser V. R., Reddy D. R.: The Hersay-II Speech Understanding System, Integrating Knowledge to Resolve Uncertainty, *Computing Surveys*, Vol. 12, No. 2, June 1980

[33] Etzion O,: PARDES - A Data-Driven Oriented Active Database Model, *ACM SIGMOD Record*, Vol. 22, No. 1, March 1993

[34] Fabret F., Regnier M., Simon E.: An Adaptive Algorithm for Incremental Evaluation of Production Rules in Databases, *Proc. 19th VLDB conf.,* Dublin 1993

[35] Fahl G., Risch T., Sköld M.: AMOS - An Architecture for Active Mediators, *Intl. Workshop on Next Generation Information Technologies and Systems (NGITS '93)* Haifa, Israel, June 1993, pp. 47-53

[36] Fishman D. et. al: Overview of the Iris DBMS, *Object-Oriented Concepts, Databases, and Applications*, ACM press, Addison-Wesley Publ. Comp., 1989

[37] Garcia-Molina H., Salem K.: Sagas, *ACM SIGMOD conf.*, 1987 pp. 249-259

[38] Gatziu S., Dittrich K. R: SAMOS: an Active Object-Oriented Database System, *IEEE Data Engineering bulletin*, Vol. 15, No. 1-4, Dec. 1992, pp. 23-26

[39] Gehani N., Jagadish H. V.: Ode as an Active Database: Constraints and Triggers, *Proc.*

17th VLDB conf., Sept. 1991

[40] Gupta A., Mumick I. S.: Improving the PF-algorithm, *Stanford rep. no. STAN-CS-93-1473*, Stanford University, 1993

[41] Hanson E. N.: Rule Condition Testing and Action Execution in Ariel, *ACM SIGMOD conf.*, 1992, pp. 49-58

[42] Hanson E. N., Widom J.: An overview of production rules in database systems, *The Knowledge Engineering Review*, vol. 8:2, 1993, pp. 121-143

[43] Harrison J. V., Dietrich S. W.: Condition Monitoring in an Active Deductive Database, Arizona State University, *ASU Technical Report* TR-91-022 (Revised), Dec. 1991

[44] Hayes-Roth D., Washington R., Hewett R., Hewett M., Seiver A.: Intelligent Monitoring and Control, *Proc. of the 1989 Intl. Joint Conf. on Artificial Intelligence*, 1989

[45] Hedberg S., Steizner M.: Knowledge Engineering Environment (KEE) System: Summary of Release 3.1, Intellicorp Inc. July 1987

[46] Ioannidis Y. E., Cha Kang Y.: Randomized Algorithms for Optimizing Large Join Queries, *ACM SIGMOD conf.,* 1990, NJ, May 23-25, pp. 312-321

[47] Katiyar A. G. D., Mumick I. S.: Maintaining Views Incrementally, *AT&T Bell Laboratories, Technical Report* 921214-19-TM, Dec. 1992

[48] Kim S-K., Chakravarthy S.: A Retrospective Analysis of Time Concepts in Temporal Databases, *Univ. of Florida, Tech. Report UF-CIS-TR-92-044*, Nov. 1992

[49] Kung H., Robinson J.: Optimistic Concurrency Control, *TODS,* 6:2, June 1981

[50] Laffey T. J., Cox P. A., Schmidt J.L., Kao S. M., Read J. Y.: Real-Time Knowledge Based Systems, *AI Magazine*, Spring 1988

[51] Litwin W., Risch T.: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering* Vol. 4, No. 6, December 1992

[52] Loborg P., Holmbom P., Sköld M., Törne A: A Model for the Execution of Task Level Specifications for Intelligent and Flexible Manufacturing Systems, *the V International Symposium on Artificial Intelligence, ISAI92*, Cancun, Mexico, Dec. 7-11, 1992, pp. 74-83 and also in *Integrated Computer-Aided Engineering*, John Wiley & Sons Inc., 1(3) pp. 185-194, 1994

[53] Lohman G. M., Lindsay B., Pirahesh H., Schiefer K. B.: Extensions to Starburst: Objects, Types, Functions and Rules, *Communications of the ACM*, oct. 1991, vol. 34, no. 10, pp. 94-109

[54] Manna Z., Pnueli A.: Models for Reactivity, *Acta Informatica*, 30:609-678, 1993

[55] Miranker D. P.: TREAT: A Better Match Algorithm for AI Production Systems, *AAAI 87 Conference on Artificial Intelligence*, Aug. 1987, pp. 42-47

[56] Morgenstern M.: Active Databases as a Paradigm for Enhanced Computing Environments, *Proc. 9th VLDB conf.,* Florence, Nov.1983

[57] Nilsson N.: Problem-Solving Methods in Artificial Intelligence, *McGraw-Hill*, New York, 1971

[58] Ohsie D., Dewan M. D., Stolfo S. J., Da Silva S.: Performance of Incremental Update in Database Rule Processing, *RIDE ADS'94*, Houston, Februari, 1994

[59] Paige R., Koenig S.: Finite Differencing of computable expressions, *ACM Trans. Prog. Lang. Syst.* 4.3 (July 1982) pp. 402-454

[60] Qian X., Wiederhold G.: Incremental Recomputation of Active Relational Expressions, *IEEE Transactions on Knowledge and Data Engineering* Vol. 3, No. 3 December 1991, pp. 337-341

[61] Risch T., Reboh R., Hart P., Duda R.: A Functional Approach to Integrating Database and Expert Systems, *Communications of the ACM*, dec. 1988, vol. 31, no. 12, pp. 1424-1437

[62] Risch T.: Monitoring Database Objects, *Proc. VLDB conf.* Amsterdam 1989

[63] Risch T., Sköld M.: Active Rules based on Object Oriented Queries, *IEEE Data Engineering bulletin*, Vol. 15, No. 1-4, Dec. 1992, pp. 27-30

[64] Rosenthal A., Chakravarthy S., Blaustein B., Blakely J.: Situation Monitoring for Active Databases, *VLDB conf.* Amsterdam, 1989

[65] Selinger P., Astrahan M. M., Chamberlin R.A., Lorie R. A., Price T.G.: Access Path Selection in a Relational Database Management System, *ACM SIGMOD conf.*, Boston, MA, June 1979, pp. 23-54

[66] Shipman D. W.: The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems*, 6(1), March 1981

[67] Snodgrass R.: Temporal Databases, *IEEE Computer*, Sept. 1986

[68] Snodgrass R.: A Relational Approach to Monitoring Complex Systems, *ACM Transactions on Computer Systems*, Vol. 6, No. 2, May 1988, pp. 157-196

[69] Stonebraker M., Row L.: The Design of POSTGRES, *ACM SIGMOD conf.*, Washington, D.C., May 1986, pp. 340-355.

[70] Stonebraker M., Agrawal R., Dayal U., Neuhold E. J., Reuter A.: DBMS Research at a Crossroads: The Vienna Update, *VLDB conf.* Dublin, Aug. 1993, pp. 688-692

[71] Ullman J. D.: Principles of Database and Knowledge-Base Systems, Volume I & II, *Computer Science Press*, 1988, 1989

[72] Widom J., Finkelstein S.J.: Set-oriented production rules in relational database system, *ACM SIGMOD conf.*, Atlantic City, New Jersey 1990, pp. 259-270

[73] Wiederhold G.: Mediators in the Architecture of Future Information Systems, *IEEE Computer*, March 1992