Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Facets of database triggers

M.H. van der Voort, M.L. Kersten

Computer Science/Department of Algorithmics and Architecture

CS-R9122 1991

Facets of Database Triggers

M.H. van der Voort
M.L. Kersten

CWI, P.O.Box 4079
1009 AB Amsterdam, The Netherlands

April 28, 1993

Abstract

Rule-based programming has found wide spread use in computer science. It is disguised as exceptions in operating systems, as production-systems and deamons in AI, as actors in programming languages, and as triggers in database systems. Generally, triggers are event-condition-action triples, where the action is executed when the condition holds after the event has occurred. In database systems triggers are used for integrity enforcement, to maintain materialized views, for notification of users, and to provide an abstraction level to the application programmer. This paper addresses three main issues of triggers in a database context: the trigger definition language, trigger semantics, and trigger optimization. The first issue focuses on the usage of triggers and on the concepts and expressiveness of trigger languages. The second issue treats several execution models, which determine the interaction of triggers and database applications. As always, optimization of triggers is of vital importance to obtain acceptable performance. Sophisticated indexing and evaluation strategies are discussed.

Categories and Subject Descriptors: H.2.1[**Information Systems**]: Logical Design- *data models*; H.2.3[**Information Systems**]: Languages- *data description languages(DDL); data manipulation languages(DML)*; H.2.4[**Information Systems**]:Systems- *Transaction processing*

General Terms: Algorithms, Documentation, Management

Additional Key Words and Phrases: Triggers, Active Databases, Rules

CONTENTS

INTRODUCTION
1. EXAMPLE
   1.1. A sample trigger definition
   1.2. A simple execution model
   1.3. Towards an efficient implementation
   1.4. Sample systems
2. TRIGGER DEFINITION LANGUAGES
   2.1. Usage of trigger systems
   2.2. Trigger languages
   2.3. Systems and literature
3. EXECUTION MODELS
   3.1. Description of execution models
   3.2. Comparison of execution models
   3.3. Single trigger execution
   3.4. Trigger activation scheduling
   3.5. Trigger cascading
   3.6. Systems and literature
4. OPTIMIZATION OF TRIGGER SYSTEMS
   4.1. Optimization strategies
   4.2. Systems and literature
5. CONCLUSIONS AND FUTURE RESEARCH
ACKNOWLEDGEMENTS
REFERENCES

INTRODUCTION

Rule-based programming has found wide spread use in computer science. It is disguised as exceptions in operating systems, as production-systems and deamons in AI, as actors in programming languages, and as triggers in database systems. Informally, triggers are operations that are automatically executed whenever a specific event occurs and a condition over a database state or state change holds. As such, triggers form the prime building blocks for active databases [Laguna Beach 1989].

Triggers have a potentially wide scope of application. An obvious application is integrity enforcement in a database system. The trigger condition can be used to capture violations of integrity rules and the corresponding action might be a corrective one, such as a transaction abort. Triggers can also be used as an interface between the database system and its environment. In that case, triggers are defined to react upon events observed from the

environment, such as the arrival of electronic mail or the failure of a remote node in a computer network. The reaction upon the former event is inclusion of the mail within a mailbox in the database. A reaction upon the latter might be an adaptation of the routing tables. Provision of an abstraction level for application programming is another use of trigger systems. Hiring a new employee could, for example, trigger the automatic assignment of a room.

Research on database triggers can be divided into three main areas: their definition, their execution models, and their optimization. The first area deals with the kinds of events, conditions and actions that can be defined. A predominant issue here is the expressiveness of the specification language. Is it sufficient to use predicate logic for the conditions and temporal first-order logic for the events? Is an SQL-like language suitable for action specification? And what language concepts do we need to communicate with the environment? A wide variety of language proposals have been published, they will be exemplified in the sequel.

The second area deals with the execution model of trigger systems. Questions to be treated are: When should the conditions be checked? Which action should be taken when multiple triggers become active at the same time? There is a great variety of articles on active databases, some specify precisely an execution model and others merely sketch an implemented system.

The third area deals with the technical problem of optimizing trigger execution. What is the best strategy for condition evaluation? How is the proper trigger to execute found? How are events administered efficiently? Techniques developed include sophisticated indexing and condition evaluation strategies.

There is a close connection between knowledge bases [Jarke, et al. 1989], [Caseau 1989] and databases with triggers, namely the similarity between rules and triggers. Both react to the state or state change of the database and infer some action to be taken. However, the prime difference between databases with rule bases and databases with triggers is their execution model. Database triggers are executed as side effects of normal database actions. Rules in knowledge bases mostly aim at information derivation and are executed upon explicit request of an application [Kiernan, et al. 1990]. This paper is primarily a survey of database triggers; an in-depth description of knowledge bases is beyond its scope.

In some systems, triggers are seen as active objects, i.e., elements of an object-oriented programming language. Therefore, they are subject to operations common to all database objects. This article refrains from surveying the object-oriented languages in-depth as a vehicle for trigger support. A recent survey is [Atkinson and Buneman 1989]. Instead, the link between the usage of triggers and the expressiveness of their definition language is discussed.

The rest of the paper is organized as follows. Each section gives a classification of problems and directions for their solution and concludes with a brief summary of relevant systems

described in the literature. Section 1 gives a short introduction to database triggers through a simple example and it gives an overview of the major trigger system prototypes developed over the last decade. These systems will be used to illustrate the principles discussed throughout the remainder of this article. Section 2 describes the major types of trigger definition languages with emphasis on their expressiveness. Section 3 presents the spectrum of trigger execution models. Section 4 classifies predominant optimization techniques applied in trigger systems. And finally, Section 5 contains our conclusions and directions for future research.

1 EXAMPLE

The concepts used in this paper are introduced by means of a full example which covers three areas of trigger systems: definition, execution model, and optimization. The example does not intend to show the usefulness of triggers but serves as an illustration of them. The first section introduces the example, the second presents its simple execution model, the third describes some simple optimization techniques to improve the efficiency of this trigger system, and the fourth introduces some existing trigger systems.

1.1 A sample trigger definition

Triggers are illustrated by means of a school administration application which consists of students and their course enrollments of the current year. The personal data of each student consists of name, address, and date of birth. The enrollment data of each student consists of class and course information. The latter is made up of the course name, a teacher and a grade.

The graduation of students at the end of the year can be described concisely with triggers, see Figure 1. These triggers are activated as soon as all grades have been given. The graduation algorithm works as follows. If the mean grade is greater or equal to 6 then the student is graduated, if it is between 5.75 and 6 the teachers make a decision, and if it is below 5.75 the student has to redo this year. For the example data shown in Table 1, Marie is automatically graduated, Jan has to redo class 6, and the teachers decide about the graduation of Karel. The graduation of Karel is subject to the *graduation2* trigger, the graduation of Jan is subject to the *graduation3* trigger, and the graduation of Marie is subject to the *graduation1* trigger. The graduation of Marie also activates the *makereport* trigger, upon the increase of the class, this trigger is activated and prints a message. All grades have to be known before a final report can be given to a student. The *submit* trigger of Figure 1 prints a warning message when there are still unknown grades at the end of June.

```
trigger graduation1
        on      update student.grade
        if      count(student.grade) = 4
        and     sum(student.grade) / 4 >= 6
        then    student.class := student.class + 1

trigger graduation2
        on      update student.grade
        if      count(student.grade) = 4
        and     5.75 <= (sum(student.grade) / 4 < 6
        then    print "subject to discussion", student.name

trigger graduation3
        on      update student.grade
        if      count(student.grade) = 4
        and     sum(student.grade) / 4 < 5.75
        then    print "redo class", student.class, student.name

trigger makereport
        on      update student.class
        if      old (student.class) < student.class
        then    print "promoted to class", student.class, student.name

trigger submit
        on      clock-tick
        if      date = "june 30"
        and     exists( student.grades contains nil )
        then    print "missing grade(s) for", student.name, student.class
```

Figure 1: The graduation and submit triggers

STUDENT						
name	**address**	**birth**	**class**	**course**	**teacher**	**grade**
Karel	street 1	12-06-1967	6	Dutch	Wetter	6
				English	de Groot	6
				Math	van Wijk	6
				Physics	de Vries	5
Marie	street 2	22-08-1966	6	Dutch	Wetter	8
				English	de Groot	8
				Math	van Wijk	7
				Physics	de Vries	9
Jan	street 3	12-06-1966	6	Dutch	Wetter	5
				English	de Groot	6
				Math	van Wijk	5
				Physics	de Vries	6

Table 1: Student data

1.2 A simple execution model

A simple execution model for the graduation example is shown in Figure 2. The picture shows five components: a data store, applications, an environment, an event history, and a trigger manager. Applications modify the database by calling the functions provided by the data store. Each call to the data store also leads to the addition of an event description to the event history. For example, each update of a student's grade leads to an entry in the history. The event history is also written by the environment; e.g. the ticking of the clock introduces history records, which allows for time-based conditions such as used in the *submit* trigger. The trigger manager reads the event history to see what events have taken place and what triggers should be activated. Once a trigger is activated, its condition is evaluated, and in case it holds, the corresponding action is executed.

Both applications and the trigger manager access the data store and the event history. To avoid conflicting updates, their execution should be synchronized. The synchronization strategy partly determines the execution model of an active database. In this simple execution model, synchronization is based on database commands. That is, after each command, control is passed to the trigger manager, which checks the event history and activates the triggers. Thereafter, the next database command is executed. When several triggers are active, an execution order must be determined. One policy is to maintain an ordering based on priority and to execute active triggers according to this ordering.

1.3 Towards an efficient implementation

The previous section describes the conceptual model for trigger execution. The actual implementation might be quite different because there are some technical problems. For instance, the event history and the trigger set may become too large to guarantee efficient

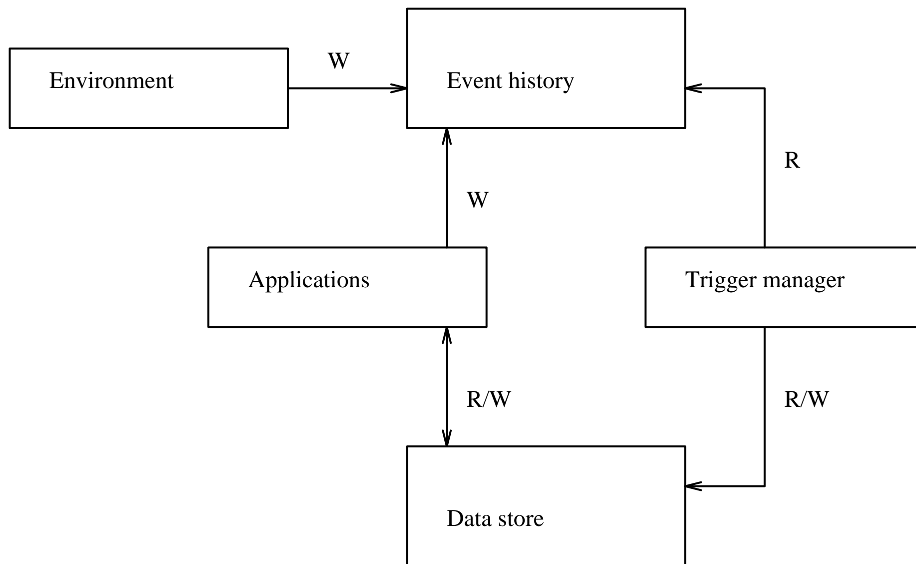

Figure 2: A simplified execution model.

system behavior. The event history can be reduced by keeping only events appearing in trigger definitions. In our example, only changes of the student relation and time events are needed. Furthermore, only the latest clock-tick and the latest update are needed, which allows for reduction of the history as well.

Using a sequential search for activated triggers is inefficient when the trigger set is large. This problem can be handled by event and condition indexing. Section 4 describes a large collection of such optimization techniques.

1.4 Sample systems

This section introduces six systems to illustrate the facets of trigger systems discussed in the remainder of this article. They are Ariel, Cactis, Hipac, POSTGRES, SQL2/3 and Taxis. These six have been chosen for several reasons. First, they are well known and documented. Second, each one explores a different part of the design space, which provides a basis for comparison.

Ariel

Ariel was developed at Wright Research and Development Center and Wright State University. It is a database management system (DBMS) with a built-in trigger system, implemented with the EXODUS database tool kit [Carey, et al. 1986]. The prime design issues are the integration of a trigger system with transaction processing and the overall efficiency of the system. The result is a conventional relational DBMS extended with a rule system [Hanson 1989].

Cactis

Cactis is an object-oriented DBMS developed at the University of Colorado. Cactis is designed to support applications that require rich data modeling capabilities, but which also demand good performance. An example is VLSI design. The system supports functionally-defined data, which can be seen as a limited form of triggers [Hudson and King 1986] [Hudson and King 1989].

Hipac

Hipac stands for HIgh Performance ACtive database system, and was primarily developed at Computer Corporation of America in Cambridge Massachusetts and the University of Wisconsin in Madison. The project addresses two problems of time constrained data management: the handling of time constraints in databases and the avoidance of wasteful polling by applications using triggers. Triggers are called situation-action rules. The semantics of the triggers are described with their knowledge model in [Dayal, et al. 1988b], and an execution model which specifies how these triggers are processed in database transactions is given in [Hsu, et al. 1988]. An implementation of these ideas and an architectural overview of the system is given in [Dayal, et al. 1988a].

POSTGRES

POSTGRES was developed at the University of California at Berkeley. It is an extended relational database management system that includes a general purpose trigger system. The trigger system is meant to be used as an implementation language for views, materialized views, partial views, procedures and caching of procedures. The design rationale of the POSTGRES trigger system is described in [Stonebraker and Rowe 1986] [Stonebraker et al. 1988] and [Stonebraker et al. 1990].

SQL

SQL is the standard database query language. The proposed extensions, SQL 2 and 3, include language facilities to describe and control triggers. Since the SQL 2 and 3 standards are language definitions rather than prototypes, they are not discussed in Section 4, which deals with implementation aspects only.

Taxis

The Taxis design language was developed at the University of Toronto. It offers an entity based framework. Some novel features are a procedure oriented exception handling mechanism and transaction definitions organized in specialization hierarchies. One of the goals of the Taxis project is to develop a theory that guides the design of semantic data models that are adequately expressive for a collection of tasks while at the same time computationally tractable. The Taxis language is described in [Mylopoulos, et al. 1980] and the design of the Taxis compiler in [Nixon 1983], [Nixon, et al. 1987] and [Chung 1984].

2 TRIGGER DEFINITION LANGUAGES

This section considers the usage of triggers and the languages for event, condition, and action definition.

2.1 Usage of trigger systems

Before we can give an in-depth discussion of the elements comprising trigger definition and its intricacies, it is necessary to instil some intuition about the usage of trigger systems in general. The usage can roughly be characterized by its abstraction level in an information system. The top level consists of triggers focussed on improved user-system interaction. This group can be divided into the following categories:

User notification: Some database situations require user intervention. Triggers can be used here to detect them automatically and to inform the user. An illustrative example is the notification of teachers to make a decision about a student's graduation.

Application procedures: Another use of triggers is the creation of an abstraction level for organizing related actions under an (external) event. Upon occurrence of the event, a series of actions is executed automatically by the trigger system. This case is illustrated by the graduation trigger: The insertion of grades gives rise to the printing action and to the graduation of some students.

Default settings: A recurring application operation is to set defaults. Triggers activated upon database insertions can be used for this purpose. For example, the insert of a student automatically sets the class to one.

Communication: Triggers offer a medium for communication. Applications can indirectly exchange data through trigger execution. An application causes a trigger to execute, which activates the target application.

The second level consists of triggers aimed at providing the functionality prescribed by a data model. This group is roughly divided into two categories:

Integrity enforcement: Integrity can be enforced with triggers. Namely, the integrity violation is specified in the condition part of a trigger and the action part specifies the corrective action for the transaction involved. For example, we could extend our example with a trigger to prohibit students born before 1950.

Protection: Database access can be controlled using triggers. In the school example, we could add a trigger that checks the user identification for all insert actions so that only teachers can assign grades.

The third level contains triggers used within the kernel of a DBMS itself. Their usage is to simplify management of physical resources and complex database states. This group can be divided into the following categories:

Physical storage optimization: Triggers can be used to adapt the storage structures, access paths, clustering, and checkpointing actions. They can also be defined to maintain statistics for query optimization. Based on such statistics, the data in our example can be re-grouped according to the classes or to the courses.

View management: Complex materialized views are easier to maintain with triggers. Whenever the base relations are updated, the materialized view can be updated by calculation of a delta change.

2.2 Trigger languages

Now that we have organized the triggers by role in the system hierarchy, it is easier to assess the event, condition, and action specification languages. In the next subsections the language concepts required are discussed in more detail.

Event and condition languages

The event and condition parts of a trigger are formulas to which a truth value is assigned upon evaluation. Consequently, logic is used as a foundation for most event and condition languages. However, the multitude of logic theories complicates the choice for the system designer. A decisive factor in the choice of a logic theory is its soundness, completeness and decidability properties. In the context of triggers, soundness means that triggers are *only* alerted in correct cases, and completeness means that triggers are *always* alerted in correct cases. A logic theory is called decidable if there exists an algorithm to decide whether a formula is derivable from the theory. This property is needed for the implementation of triggers which partly consists of the evaluation of (event) conditions. A sound and complete theory does not guarantee the existence of such an algorithm.

Below we give a classification of possible logic theories for event and condition languages. For each logic, we first give the alphabet used to construct formulas. Then a possible interpretation of these formulas in a database context is given and the soundness, completeness and decidability of the logic are discussed. For a more general treatment of logics and their interpretation in a database context see [Gabbay and Guenthner 1983] [Frost 1986] [Kifer and Lozinskii 1989] [Small 1986].

Propositional logic: The propositional vocabulary consist of propositions and connectives. Propositions are statements which are either true or false, the connectives

are negation, conjunction, disjunction, implication, and equivalence. Formulas are constructed from propositions and connectives.

For databases, the objects can be seen as propositions. Objects in the database are assigned the value true, tuples not in the database are assigned the value false. Only simple conditions on database values can be expressed, such as "Marie's salary is 500 dollars". The condition "all salaries should be 500 dollars" can not be expressed without enumerating all the occurrences, because the propositional logic does not support quantified variables.

The propositional logic is sound, complete and decidable. Despite these nice properties, it is of limited use as an event or condition language due to its limited expressiveness, i.e. lack of variables and quantification over the database contents.

First order logic (predicate logic)**:** The basic symbols of a common predicate logic are functional constants, predicate constants, variables, connectives, the universal quantifier, and the existential quantifier. The elementary parts of a predicate formula are predicate constants and expressions formed with functional constants and predicate constants. The difference between the propositional logic and the predicate logic are the variables and quantifiers.

In the database context, the predicates are associated with database objects. A predicate is true if the associated object is in the database, otherwise it is false. Variables make it possible to express conditions like "all salaries are 500 dollars".

The first order logic is sound and complete, but not decidable. It is possible to define triggers that are never executed although their event and condition should evaluate to true. The undecidability of the first order logic makes it unsuitable for triggers.

Horn logic: A logic with expressiveness between propositional and first order logic is Horn logic. The formulas of Horn logic are restricted to a conjunction with at most one negated disjunction. of which at most one is a positive literal.

Horn logic is sound, complete and decidable. For this reason it is suitable for expressing database state conditions. However, events specify database transitions that need the notion of time e.g., every five minutes do. Horn logic lacks temporal operators, and so is not directly suitable for event specification.

Temporal logic: The aforementioned logics lack the facilities to express conditions over time, i.e. conditions over state changes can not be specified. This has lead to a branch of logic, called temporal logic. Temporal logic is used to reason about program behavior in time. In our case the programs are the applications and the database represents the variables manipulated by applications. To reason about its progressive states, two operators are added to a logic, the *necessarily* ($\Box$) and the *possibly* ($\Diamond$) operators. Let F be a formula. $\Box$(F) means that F is always true and $\Diamond$(F) means that F will be true at least once. For more information see [Gabbay and Guenthner 1983].

Within these logics it is possible to express formulas like "Jan moved from Amsterdam to Rotterdam this year".

A proposition or Horn logic system extended with temporal operators is still sound, complete and decidable. Therefore, a dynamic Horn logic can be used for event definition. Some examples are given in [Cohen 1989] and [McCarthy and Dayal 1989].

Higher order logic: Higher order logic allows for simple specification of complex conditions. One of its features are variables ranging over predicates. However, it is incomplete and undecidable. Therefore, we do not consider it in the remainder of this report.

Action languages

The action language of triggers can be divided into three groups; query languages, extended query languages, and languages which offer direct database access.

Query language: The action language is the same as the query language of the database system. This results in triggers that have internal effects, i.e. the triggers can only affect the database. An example action is "insert into student". The prime advantage of this choice is that access to the database is under control of the database system itself. A disadvantage is their limited functionality. No actions on the external environment can be defined.

Extended query language: Triggers interacting with their external environment often require extensions to the native query language of the DBMS. For example, a mail system based on triggers needs facilities for external communication. An action language based on two parts, a query language and a communication language, combines controlled database access with communication facilities. The communication can be based on function calls or on send and receive primitives. The action "update student and send message" is an example of a communicating action.

Direct database access: Providing a complete algorithmic language greatly improves the expressiveness of the action language. The action "recluster the student data" is an example of this. This action cannot be expressed in an extended query language. The disadvantage, however, is the reduction of optimization options. Programs written in a restricted and type secure language are simpler to analyse than programs written in an untyped language.

Table 2 shows the relationship between the action languages and the main usage for triggers. User notification triggers need one way communication with users. If a query language is used, only query answers can be used to inform users. This means that all possible messages should be stored in the database, which is not as flexible as a printing routine where the

message is an argument. Thus, the query language can be used but the extended query and the direct database language are better suited for user notification triggers. Default setting is the assignment of values to tuples, which is a part of all query languages. Therefore, it can be realized with languages of all three classes. The more expressive the action language, the better application procedure and communication triggers are supported. For example, a query language restricts communication to the exchange of data via database relations, while direct communication is offered by an extended query language. The enforcement of integrity and protection is based on the ability to revoke actions, which is not possible in a query language. Therefore the query language is not suited for protection and integrity enforcement. Data storage management requires the ability to allocate memory, to write data to disk, etc. Therefore, a direct database access language is needed.
The efficient construction and management of materialized views also require memory control. Thus direct database access is needed.

An extended query language suffices for most trigger usage. Only low-level usage can not be supported with such a language, they need a direct database language.

	query	**ext. query**	**direct db access**
user notification	+	++	++
application procedure	+	++	++
default setting	++	++	++
communication	+	++	++
integrity enf.	-	+	++
protection	-	+	++
physical stor. opt.	-	-	++
view management	+	+	++
the meaning of the symbols ++ can be efficiently expressed + can be expressed - can not be expressed			

Table 2: Action languages for trigger support

2.3 Systems and literature

In this section, we discuss the trigger definition languages of our sample systems. For all systems, we give an overview of the syntax for trigger definition and we describe the systems' main goal.

Ariel

Ariel is based on the relational data model. It uses a subset of the POSTQUEL query language [Stonebraker et al. 1988]. The Ariel trigger language (ARL) is a trigger language designed for database environments. In particular, trigger conditions in Ariel can be based on a combination of database events and pattern matching over event sequences. Ariel trigger conditions can also test for transition conditions. These transition conditions have

a truth value which is a function of the database state before and after a transition. For example, *previous student* refers to the previous state of the student relation (i.e., the state of the relation at the beginning of the current transaction). The general syntax of an Ariel trigger is

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

The *priority* clause provides precise control over the execution order of triggers (see Section 3). The *on* clause allows specification of an event that will activate the trigger. Two kinds of events are supported: time events and database events such as append, delete, replace, and retrieve. The *if* clause constrains trigger firing to those cases where conditions on the current and the previous database state are true. The Ariel operator *previous* is used to support temporal conditions. The operator refers to the previous state of its argument. The conditions are based on Horn logic. The *then* clause specifies the actions to be executed when the trigger becomes activated. An action is a sequence of database statements. Ariel does not support aggregates in triggers. Therefore, the graduation triggers from our example can not be expressed in Ariel since they use the *sum* aggregate.

Cactis

The data model underlying Cactis is based on a principle called active semantics. It is designed to support complex functionally-defined data. In an active semantic database, each object has a behavioral specification and can respond to changes occurring elsewhere in the database. Cactis objects have the following syntax:

object type *object-name*
[**relationships** *dependency-list*]
[**attributes** *attr-constraint-list*]
[**rules** *rule-list*]
end

Each object may be a piece of either stored data or derived data (i.e., data that depends on other objects), and it may have associated constraints. The system automatically maintains derived data and enforces constraints. The automatic maintenance of derived data is a restricted form of triggers. Whenever data changes (i.e., an event occurs), the associated constraints are checked (i.e., condition evaluation), and the derived data is updated (i.e., an action is taken). Alternatively, upon constraint violation, the transaction causing the violation fails and is rolled back. Constraints are specified using Horn logic. The actions are applicative and expressed in a computationally complete data language.

Hipac

Hipac proposes Event-Condition-Action (E-C-A) triggers, called rules, as a formalism for active database facilities. Trigger usage such as application support, implementation of data model functionality, and implementation of the database management system itself can be realized with these general triggers. Hipac extends the PDM data model [Manola and Dayal 1986] to include triggers as first-class objects. The PDM data model integrates the functional, relational and object-oriented data models. The Hipac trigger syntax looks like:

> *rule-identifier*
> **event** *event*
> **condition:**
>   **coupling:** *coupling-mode*
>   **query:** *query*
> **action**:
>   **coupling:** *coupling-mode*
>   **operation:** *operation*
> [**timing-constraints** *constraint-list*]
> [**contingency-plans** *contingency-plans*]

There are three kinds of primitive events: data manipulation, the external clock, and external notification. Hipac supports recognition of composite events using three event constructors: the disjunction of two events, the sequence of two events, and the closure of an event. A composite event is a regular expression built with these constructors. The motivation for the closure operator is that often a trigger should be fired once per transaction, provided a given event was signaled *at least* once during a transaction, rather than firing every time the event was signaled. The closure of an event E is signaled after E has occurred an arbitrary number of times in a transaction.

The condition of a trigger is a collection of queries. The collection is said to be satisfied if all queries return non-empty answers. The main reason for using a collection of queries rather than a simple predicate is that they provide the flexibility of passing additional arguments to the action part, namely, the results of the queries. Coupling modes describe the link between event occurrence, condition evaluation and action execution. They are discussed in the next section. The condition is based on Horn logic and the event is based on Horn logic extended with temporal operators.

The action for a trigger can be a program written in the data manipulation language of the system, or it can be a message to an external program or process. The timing constraints specify how long an action can take. The contingency plans specify alternative actions in case the time limit is exceeded.

POSTGRES

The first version of POSTGRES [Stonebraker et al. 1988] used a trigger activation paradigm

in which a command was logically *always* or *never* in execution. The second version of the POSTGRES [Stonebraker et al. 1990] trigger system takes a more traditional production-system approach. The syntax of a POSTGRES trigger looks much like the triggers in other systems. The general form of its triggers is:

> **define rule** *rule-name* [**as exception to** *rule-name*]
> **on** *event* **to** *object* [[**from** clause] **where** *clause*]
> **then do** [**instead**] *action*

The event is one of the following: *retrieve*, *replace*, *delete*, *append*, *new* and *old*. *New* stands for (*replace or append*) and *old* stands for (*delete or replace*). The *where* clause is comparable to a standard database query condition which is a Horn logic formula. The action part is a collection of POSTQUEL commands with the addition of the predefined variables *new* and *current*. POSTQUEL commands are database queries like retrieve, append and delete. These variables contain the values of the changed object. *Current* refers to the object value before the event occurred and *new* refers to the object value after the event occurred.
The event is a single predicate and the *where* clause is based on Horn logic.

SQL2/3

Triggers in SQL2/3 are event-condition-action triples. They are meant to be used for application support. The general trigger syntax is:

> **create trigger** *trigger-name*
> *trigger-event* **on** *table-name*
> **when** *search-condition statement-list*

The event types recognized are *insert*, *delete*, and *update*; these are the common data manipulation statements of SQL2/3. The condition part is a common SQL2/3 predicate (i.e., a Horn logic formula) and the action part is a sequence of SQL2/3 data manipulation statements. As in POSTGRES the action part may use the predefined variables *new* and *old*.

Taxis

Taxis is designed primarily for application systems that are highly interactive, make substantial use of a database, and have a predictable interaction structure. The latter leads to the concept of a transaction-class, which consists of a parameter-list, locals, pre-requisites, actions, and post-requisites. The combination of pre- and post-requisites with action execution is equivalent to triggers. Their main usage is the enforcement of integrity. The general transaction syntax is:

> **define AnyTransactionClass** *name parameter-list* **with**
> **prerequisites** *prereq-list*
> **actions** *action-list*

postrequisites *postreq-list*
EndAnyTransactionClass

The description language of the pre- and post-requisites is based on Horn logic. The pre- and post-requisites should hold for the transaction execution to be meaningful. If they do not, an exception is raised and an exception handling transaction is called upon to correct the situation. The nature of actions is a mixture of database (i.e., insert) and programming language concepts such as conditional, block and looping constructs.

3 EXECUTION MODELS

The previous section described the facets of trigger definition languages and showed their usage in actual systems. This section discusses trigger execution models. The existence of triggers in a database system affect its execution model significantly. A traditional database is a passive repository. The inclusion of triggers turns it into an active one because triggers are executed automatically by the system. In this section, we discuss various execution models. First, we introduce some basic concepts to simplify the description of models encountered in the literature. Second, the predominant execution models are described and compared. And third, the execution models employed in our sample systems are illustrated. Another comparison of execution models is given in [Zertuche and Buchmann 1990].

3.1 Description of execution models

In our simple execution model (Section 1), the graduation triggers are executed after each update of the student grades. Conceptually, after each database command, the system checks for triggers eligible for execution. If there are any, they are executed as atomic actions. This simple example shows some of the main concepts encountered in execution models: the application granularity, the trigger granularity, and the Transaction/Application (TA) schedule.

The *application granularity* is the level at which trigger activation is detected. In our example, it is the simple database command, such as the update of a grade. After each database command, the system checks whether there are activated triggers and executes them. Figure 3 shows some other options for choosing the application granularity: a database session, a database transaction, and a database operation (e.g. tuple level read or write). Box A1 of Figure 3 shows a whole session as the application granularity. Trigger activation can only be detected between sessions, not between intra-session units of activity (e.g., transactions). From left to right, the granularity becomes finer, with the simple database operation being the finest.

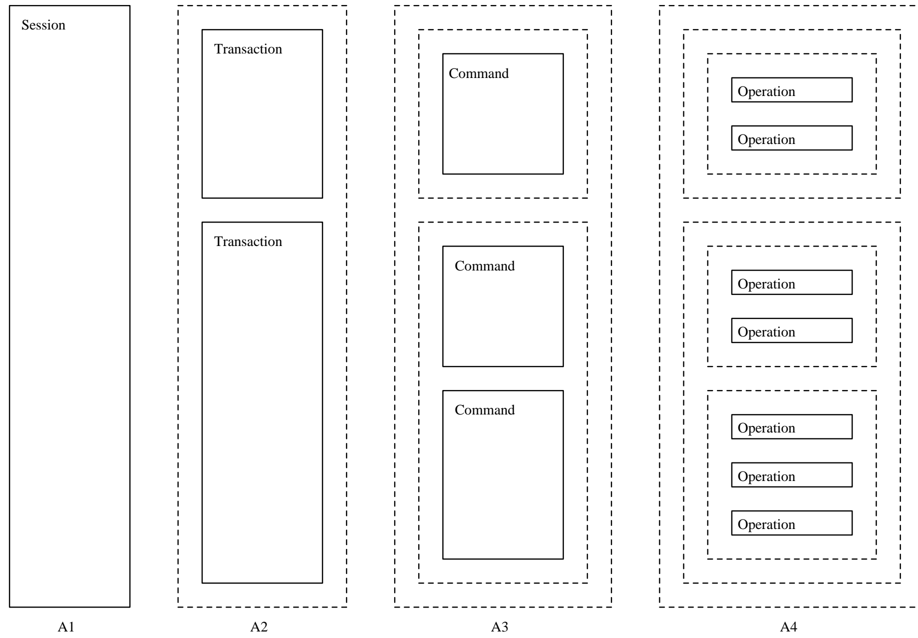

Figure 3: The application granularity.

The *trigger granularity* is the level of atomicity at which the trigger components are executed. In our school example, it is the complete trigger, i.e., the event checking, condition evaluation, and action execution are not interrupted by the execution of the application. Figure 4 also shows some other possibilities. In T2, the event, condition, and action are individually atomic; In T3, the low-level operations that constitute the event, condition and action are atomic. As in Figure 3, the granularity becomes finer from left to right.

Consider a trigger system based on application granularity A2, i.e., a database transaction Tr_i, and trigger granularity T2, i.e., a trigger split into its three basic components E_i, C_i and A_i. Then for an application run

$$AG = \{Tr_1, Tr_2, ..Tr_n\}$$

and for trigger execution

$$TG = \{E_1, C_1, A_1, E_2, C_2, A_2, ...E_m, C_m, A_m\}$$

The outcome of the system is determined by a schedule over AG and TG (*TA schedule*). An example schedule is

$$< Tr_1, E_1, C_1, Tr_2, Tr_3, A_1, E_2, C_2, A_2, ... >$$

Clearly, only meaningful schedules should be generated and the semantics of such schedules should be defined. In this example, we assumed that Tr_1 and Tr_2 are independent and that trigger 1 broken into smaller components, is not affected by the sequence $< Tr_2, Tr_3 >$. The semantics of these schedules can be defined by extending the conventional transaction semantics. Therefore, the unit of execution established by the trigger granularity is usually interpreted as a subtransaction [Moss 1981] and [Pu 1986].

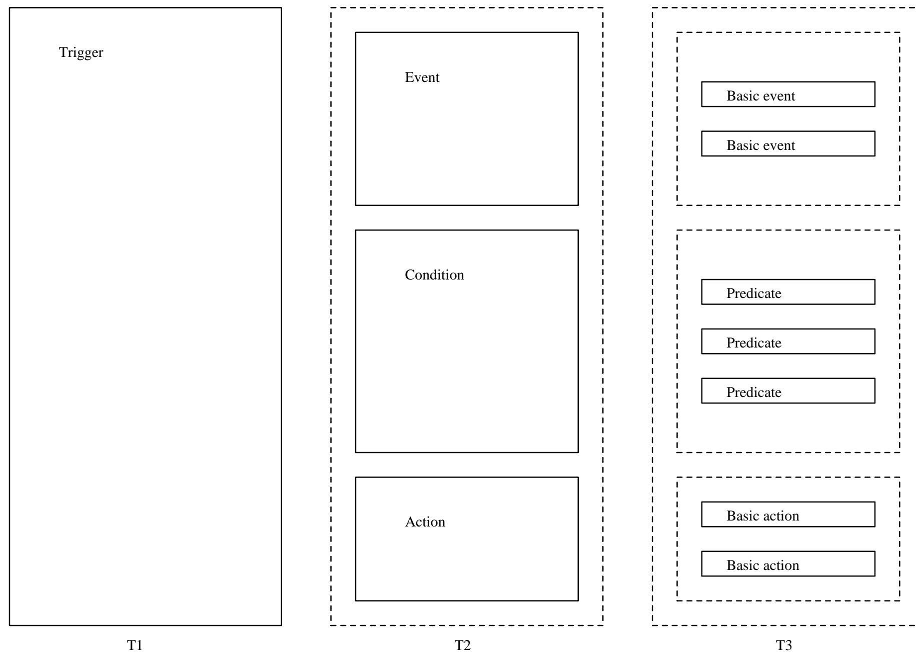

Figure 4: The trigger granularity.

3.2 Comparison of execution models

The granularities used for application and trigger interleaving leads to a two-dimensional space of execution models. An execution model has repercussions on the trigger uses supported by a trigger system. This subsection compares the execution models based on the trigger usages mentioned in Section 2. First, all usages and the requirements they pose on the execution model are described. Second, this information is compactly presented in Table 3 and some additional comments are made.

Triggers for user notification, application procedures, communication, and storage optimization are supported in all execution models. However, a finer application granularity leads to more possibilities for these trigger usages. If the application granularity is the database session, then messages can only be sent after the session has been finished, which severely limits interaction with the user. However, if the application granularity is a database command, then messages can be sent after each command and it is possible to define triggers which clearly interact with the user. Therefore, the granularity A3 is better suited for these triggers than granularity A1.

Protection and integrity enforcement in a database context is part of the transaction semantics. Conceptually, enforcement takes place just before a transaction is committed. Using triggers for the implementation of protection and integrity enforcement requires control at database command level; triggers should be executed just before a transaction commits. Control at transaction level is not sufficient, because triggers can only be executed after a

transaction commits, which is obviously too late.

A database command may use the result of preceding commands. Therefore, default setting and view management triggers require database access at command level.

Table 3 gives an overview of which usage is supported by which class of execution models. The rows show the usage and the columns the classes. The execution model classes are based on two dimensions: the application and trigger granularity. However, the mentioned use of triggers only requires class formation based on the application granularity. Therefore, the columns of Table 3 show classes based on the application granularity. All mentioned trigger usage is specified at database level. The lowest database level is the database command level. Therfore, the application granularity A4 is not needed, all usage can be supported with control at database command level. Consequently, the A4 granularity is left out of Table 3. No indication of the required TA sequence is given in Table 3, the requirements are mentioned in the above given description only.

Trigger usage	Application granularity		
	A1	**A2**	**A3**
user notification	+	++	+++
application procedures	+	++	+++
default setting	-	-	+++
inter-operability	+	++	+++
integrity enforcement	-	-	+++
protection	-	-	+++
storage optimization	+	++	+++
view management	-	-	+++
the meaning of the symbols +++ is supported very well ++ is supported good + is supported - is not supported			

Table 3: Goals supported by execution model classes.

3.3 Single trigger execution

During event and condition evaluation, variables are bound to objects in the database. These bindings are used in the action part of a trigger. For example, consider the graduation triggers of Section 1. In the **on** and **if** clause the variable *student* is bound; this binding

is used in the action part for printing the name of the student. There are two kinds of variables: user defined variables such as *student* and system defined variables such as *new* in POSTGRES. The set of all objects that satisfy the event and condition is called the *binding set.* Often, the single trigger semantics is described with the use of the binding set. There are two options for single trigger execution.

option 1: The action is executed once for each object in the binding set. These action executions are all part of the same trigger execution [McCarthy and Dayal 1989] [Cohen 1989].

option 2: The action is be executed once for the whole binding set. The binding set is assigned to a predefined system variable which can be used in the action part [Widom and Finkelstein 1990] .

See Figure 5 for a pictorial representation of the policies. The first option is used in our school example, where the variable student refers to a single tuple of the student table. For example, the action of the makereport trigger is executed for each student that is advanced to a higher class.

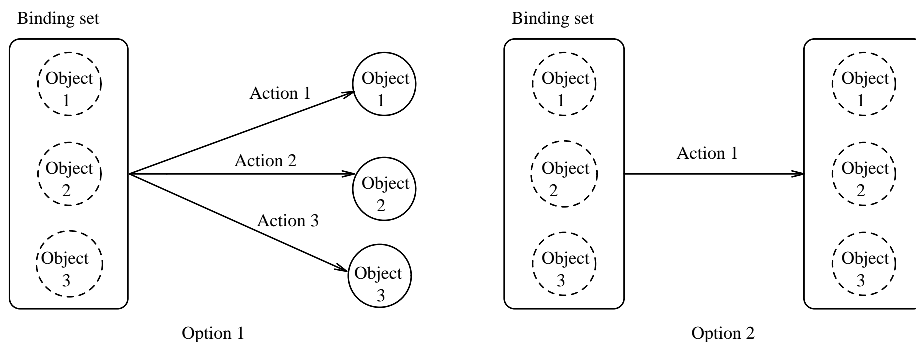

Figure 5: Single trigger execution. A binding set consisting of three elements results with option 1 in three action executions and with option 2 in one action execution.

3.4 Trigger activation scheduling

Most systems allow the user to specify triggers with overlapping conditions. Therefore, multiple triggers might become active at the same time. Such a group is called the *active trigger set.* A trigger set with more than one trigger leads to the problem of inconsistent system behavior. Several algorithms have been proposed to realize consistent behavior. Before we discuss them in detail, we introduce some common terminology.

Triggers *interfere* with each other when their action parts access the same data. When they all only read data, they do not interfere. An example of an interfering trigger pair is: trigger A reads tuples from relation R and trigger B writes tuples to relation R, and

both are activated upon an update of relation R. The outcome of these triggers depends on their execution order. If trigger B is executed before trigger A, the tuples written by B can be read by A. If trigger A is executed before trigger B, the tuples cannot be read. There are two strategies to achieve consistent behavior: pick only one trigger to execute or fix an execution order e.g., alphabetically ordered by trigger name.
A trigger T1 is called a *specialization* of trigger T2 if the activation of T1 implies that T2 is activated at the same time too. For example, the condition *A=1 and B=1* is a specialization of *A=1*.
The following list describes some execution strategies for the active trigger set.

Parallel execution: All triggers are executed in parallel. The outcome of this strategy is only predictable when the triggers do not interfere.

Ordered sequential execution: All triggers are executed sequentially, however their execution order is chosen either at random, or by priority (given in the trigger definition), or based on a cost function (e.g., the least cost). When the triggers do not interfere, the effect of a random execution order is the same as parallel execution. Otherwise it results in unpredictable system behavior. Priority-based selection offers the application the possibility to partly control the execution order. This might lead more predictable outcomes. The function-based selection is more flexible then priority based selection. At run-time, the function is computed and a trigger is selected. Again, the client determines the execution order, this time by supplying the function. The execution of all triggers instead of just one seems to be sensible when they have no specialization relation. When they do have a specialization relation, only the most specialized activated trigger should be executed.

single trigger execution: One trigger is executed. It can be chosen at random, by priority, or based on a function. In general, this option is useful when there exists a strict ordering based on the specialization of the triggers.

Tree ordered execution: The triggers are grouped into specialization trees. From each tree, the most specialized, activated trigger is selected for execution. The triggers activated for each binding (section 3.3) are seen as trees of one element. The result of this strategy is not always consistent, it depends on the interference of the executed triggers.

To use triggers correctly, their execution should have a predictable result. Therefore, parallel execution should primarily be seen as an optimization technique. Tree ordered trigger execution is a combination of ordered sequential and single trigger execution. The selection of triggers from the trees is done as in the single trigger strategy, and their execution is the ordered sequential strategy. The advantage of tree ordered execution is its modular structure, which simplifies the design of complex trigger systems.

One of the problems of non-singleton active trigger sets is the possibility of producing inconsistent results. For example, trigger T1 might insert an object which is deleted by trigger T2. Therefore, tools are required to detect inconsistent triggers at design time and to guide the application in the definition of triggers. Some work on this important research issue is described in [Ioannidis and Sellis 1989] and [Zhou and Hsu 1990].

3.5 Trigger cascading

The execution of a trigger body might also activate another trigger, which is called cascading. For example, insertion of a new student results in the default insertion of the subjects required for graduation, which in turn, results in an update of the enrollment statistics.

The advantage of trigger cascading is a uniform treatment of all (sub)transactions, including the ones constituted of triggers. However, a prime disadvantage is the possibly opaque system behavior and the sensitivity for livelock situations. In particular, cascading may lead to cyclic trigger activations. Therefore, convergence criteria are needed to detect such situations

3.6 Systems and literature

In this section we review the execution models of the six prototype systems. Their granularities, the TA schedule, single trigger semantics, cascading, and the multiple trigger execution strategies will be discussed.

Ariel

Ariel triggers can be defined to react to simple database statements like delete and replace. So the events are detected at command level which is the application granularity A3. However, the action is executed at the end of the top-level command in which the event occurs. Ariel defines a *compound command* to be a **do .. end** block surrounding a list of other commands. A command is called *top-level* if it is not nested inside a compound command. This is an example of split trigger execution, namely, the T2 granularity in which the event and condition evaluation is separated from the action execution. A trigger action is executed once for each object in the binding set and trigger execution is part of the transaction in which the trigger is activated. This means that if the trigger action fails, the transaction is aborted.

Trigger selection is based on priority, time of activation, and selectivity of the trigger. The selectivity depends on the number of accessed objects and is estimated by the query optimizer at trigger compile time. If this does not result in a single trigger to be executed, one is arbitrarily selected. Trigger cascading is not explicitly forbidden, it is also not directly supported.

The trigger has access to both the database state and to the variables set by the activating event. For example, if the event is **append to** R then the variable R is bound to the set of tuples just appended to R. This means that statements executed between the event/condition evaluation and the action execution do affect the database state seen by the action, but they do not affect the variables set. It is possible that the variable contains tuples which are no longer in the database by the time the action is executed. This might result in odd situations. For example, the action part of the trigger updates tuples which are no longer in the database. This eventuality is not considered in Ariel.

Cactis

In Cactis derived attributes are updated directly after a change of its dependent attribute. Non-derived attributes are changed by database commands. Thus the application granularity is the database statement (A3) and the trigger granularity is the complete trigger (T1).
Multiple triggers and cascading are supported. Cactis differentiates between intrinsic and derived attributes. Intrinsic attributes do not depend on other attributes, whereas derived attributes do. Starting from the intrinsic attributes, a dependency graph can be constructed that describes the dependencies of attributes. The recalculation of attributes which depend on derived attributes can be seen as a form of multiple triggers. The recalculation of the whole dependency graph is trigger cascading.

Hipac

One of the design goals of Hipac is to provide timely response to critical events. It may be important to evaluate the condition immediately after the event has occurred and to execute the action part immediately thereafter. To achieve this goal, the Hipac triggers are extended with two kinds of coupling modes: E-C and C-A. E-C coupling specifies when the condition is evaluated relative to the transaction in which the triggering event is signaled. C-A coupling specifies when the action is executed relative to the transaction in which the condition is evaluated. There are three kinds of coupling strengths: *immediate*, *deferred*, and *detached*. In *immediate* E-C mode, the condition is evaluated immediately after the event occurs. In *deferred* E-C mode the condition is evaluated in the same transaction in which the triggering event occurred, but just before transaction commits. In *detached* E-C mode the condition is evaluated in a separate transaction. The interpretation of the C-A mode is similar.

Thus, the trigger granularity is determined by the coupling modes. It varies between the complete trigger (T1) and the trigger split into event, condition, and action parts (T2). The application granularity is the database operation in Hipac. However, a database operation ranges from single tuple read/write to groupings of multiple, more complex operations.

If an event activates more than one trigger, then for each condition-action pair a separate subtransaction is created. There is no conflict resolution policy that chooses one trigger to fire. Instead, all triggers fire concurrently. Hipac assumes the triggers to be free of contradictions, which is up to the application to guarantee. They make the same assumption

with respect to cascading.

The action is executed once for each object in the binding set. These objects are passed to the action through the arguments of the action. The action may also refer to the database state at the current time, i.e. the time the operation is being executed, and to the database state before the event took place. The free variables are bound to the set of objects satisfying the event and condition, and the action is executed once for this set.

POSTGRES

The execution model of the POSTGRES trigger system is defined at tuple level. At the time an individual tuple is accessed, updated, inserted, or deleted, there is a *current* tuple (for retrieves, replaces and deletes) and a *new* tuple (for replaces and appends). If the event and the condition specified in the *on* clause are true for the *current* tuple, then the action part of the trigger is executed. Thus the application granularity is a database operation (A4) and the trigger granularity is the complete trigger (T1). Cascading and multiple triggers are not described for POSTGRES.

SQL2/3

The execution model for a trigger is described in the trigger definition by tags. A tag specifies when the trigger should be executed, i.e., before or after the event. The before-or-after tag guides the construction of the TA schedule. Another tag specifies how often the action part should be executed: just once or once for every tuple in the result of the trigger condition.

The scheduling of triggers with transactions is not explicitly described. Triggers seem to be part of the activating event which is a database command and, therefore, to be part of the same transaction as the activating event. This results in the database command as application granularity (A3) and the complete trigger as trigger granularity (T1).

To avoid difficulties, cascading and conflicting triggers are not allowed. The latter is established by disallowing more than one trigger definition in which the same event is specified. How the former is enforced is not described.

Taxis

The combination of transaction and trigger definition in Taxis clearly indicates the close link between them. Triggers are executed at the beginning and at the end of a transaction. Only the triggers (i.e., pre- and post-requisites) defined for the transaction are executed. The application granularity is the database statement(A3), the trigger granularity is the complete trigger(T1), and the TA schedule results in the execution of a trigger at the beginning and end of a transaction.

The Taxis *transaction class* concept prohibits the occurrence of multiple active triggers. Only two can be specified; one to be executed at the begin of a transaction and one to be executed at the end. Trigger cascading does not exists in this model.

The arguments of the transaction class may be used everywhere in the transaction and they are bound in the pre-requisite part. The action is executed once for each possible binding.

Miscellaneous

A detailed description of an execution model is given in [Widom and Finkelstein 1990]. Their trigger execution semantics is based on execution blocks and affected sets. In our terminology, the execution block corresponds with the application granularity. In the paper, execution blocks are assumed to be transactions (T1). The affected set describes the changes to the database after the execution of a block. Triggers are part of the activating transaction and are allowed to (de)activate other triggers. The authors do not address concurrent execution of multiple triggers, so the triggers are executed in some unspecified serial order.

[Cohen 1989] describes an execution model for constraints. The constraints refer to observable states, i.e., states that should hold at the end of a transaction.

4 OPTIMIZATION OF TRIGGER SYSTEMS

This section deals with a prime implementation problem of trigger systems, namely, their optimization. Due to the objectives of different trigger systems and the diversity of underlying system architectures, there exist many optimization techniques. We do not discuss implementations in detail, but merely describe their main characteristics.

4.1 Optimization strategies

In a nutshell, optimization techniques can be grouped into three categories: optimization of query execution, reduction of query execution, and optimization of storage.

The first group consists of algorithms to optimize the query execution plan. They are common database query optimization strategies [Ullman 1980].

Perfom selections as early as possible: This tends to decrease the intermediate results of queries.

Preprocess storage structures: Sorting and indexing of data reduces the look-up time.

Recognition of common subexpressions: The conditions of triggers are scanned for common subexpressions. Recognizing common subexpressions allows them to be evaluated just once.

Combination of operations: The combination of operations reduces the number data scans.

Some examples of their use in trigger systems is given in [Stemple, et al. 1987] [Cohen 1989] [Hsu and Imielinski 1985] and [Small 1986].

The second group consists of algorithms to reduce the amount of computation. These algorithms try to avoid redundant computation of common subexpressions and unnecessary trigger execution.

Incremental evaluation of conditions: Instead of a complete evaluation of conditions, they can be evaluated incrementally. Upon a possible change of the truth of a conditions value, only the affected subexpressions of the condition are re-evaluated. To make this approach work, conditions are parsed and organized into a condition tree. The leaves of this tree are simple predicates. Each node of the tree contains the truth value of the corresponding subexpression. Database actions and external events may change the truth value of the leaves. Only the truth values depending on those leaves are recalculated. The other truth values are still valid and need not be recalculated. [Sun, et al. 1989] [Chakravarthy and Minker 1986] [Rosenthal, et al. 1989].

Early/lazy trigger execution: The preceding strategies optimize condition evaluation. Another track is the reduction of computation time involved in handling the action part. This can be achieved by early execution of triggers, where they are executed as soon as possible, or by lazy execution where triggers are executed upon a request for the result of the execution.

Early execution can be beneficial when a trigger is used for integrity enforcement. Instead of executing the trigger at the end of a transaction the trigger should be executed as soon as a constraint might be violated. The early recognition of constraint violation leads to an early transaction abort [Bertino and Musto 1988]. Late execution is beneficial when the trigger updates the database more often than the result is needed.

The third group consists of strategies for storage optimization of both data and triggers. Some optimize the look-up time for data and triggers and others reduce the amount od stored data.

Data indexing: Database objects are grouped according to predicates. Triggers have object groups assigned to them in such a way that only objects in these groups have to be considered for condition evaluation [Hanson, et al. 1990], [Cohen 1989], [Forgy 1982], [Risch 1989], [Chakravarthy and Nesson 1990] and [Nicolas 1982].

Trigger indexing: The triggers are indexed according to the events that might activate them. Once an event occurs, possibly activated triggers can be found quickly. [Antunes, et al. 1990].

Finite automata: The events and conditions of all triggers are represented by a finite automaton. The state of this automaton represents the truth value of the trigger

conditions. Database actions and external events result in a state change of the automaton. The advantage of this representation is a reduction of data. Only a representation of the automaton has to be stored instead of storing the complete history of all objects. This is especially important when temporal conditions are used, because they need the history of objects [Hulsmann and Saake 1990].

4.2 Systems and literature

This section describes the optimization strategies used in the sample systems. Unfortunately, most papers on trigger systems do not explicitly discuss the optimization techniques being used. Information about optimization techniques can mostly be found in literature on query optimization such as [Bancilhon and Ramakrishan 1986] [Graefe and DeWitt 1987].

Ariel

The event and condition evaluation of Ariel is based on the Rete algorithm of [Forgy 1982]. This algorithm incrementally maintains sets of objects satisfying conditions. In case an event activates a trigger, all objects that satisfy the trigger condition can be found in a maintained set.

Cactis

Cactis uses a twofold optimization strategy: reduction and deferment of computation. The first strategy evaluates derived attributes according to an algorithm which computes only those attributes whose value changed as a result of a database modification. It avoids needless recomputations. This evaluation algorithm first determines what work has to be done, and then performs the actual computations. For this, the algorithm uses the dependencies between the attributes. When the value of an attribute is changed, it may cause the dependent attributes to become out of date. Instead of immediately recomputing these values, they are simply marked as out-of-date. This marking process is recursively applied until no new out-of-date attributes are found. Then the out-of-date attributes are evaluated.

The second strategy defers the calculation of attribute values that are not directly needed. If the user explicitly requests the value of those attributes, new computations are made to obtain the current values.

Hipac

One of the prime concerns of the Hipac project is efficiency. Therefore, they have identified a range of techniques in order to optimize the evaluation of complex conditions. Amongst them are multiple condition optimization, derived data management, and incremental condition evaluation. They are not yet implemented. They use a Rete network-like structure [Forgy 1982] for condition processing which is a form of data indexing.

POSTGRES

To optimize trigger execution, two different implementation strategies are used in POST-

GRES. The first is through tuple-level execution, the second is through a query rewrite. The tuple-level implementation maintains event locks at tuples. These locks are placed on appropriate fields in all tuples of the relations that appear in the event specification of a trigger. These relations must satisfy the event qualification. Event locks are tagged with the type of event that will activate the action. When an event occurs on a field marked by an appropriate event lock, the qualification in the trigger is checked, and if true, the action is executed. During query processing, it can immediately be seen which trigger should be executed. Query rewriting takes place between parsing and optimization. The query rewriting component compiles commands together with the triggers which apply to them, thus avoiding dynamic trigger searches.
Depending on the trigger, one of the implementations is chosen. Intuitively, if the event accesses most of a relation, then the rewrite implementation will probably be more efficient; if only a small number of tuples are involved, then the tuple level system may be preferred. Specifically, if there is no *where* clause in the trigger event, then the query rewrite will always be the preferred option.

Further optimization is achieved by postponing the execution of triggers or by evaluating the action statements in advance. See [Stonebraker et al. 1990] for additional details.

Taxis

Taxis transactions are meant to be executed several times; therefore, compilation should be efficient. It is similar to the query rewrite of POSTGRES. The compilation of triggers together with transactions avoids the search for activated triggers after transaction execution, since they are already identified at compile time.

5 CONCLUSIONS AND FUTURE RESEARCH

We have presented a survey on database triggers covering both theory and system aspects. Triggers can be used as a modeling tool by users for application development, an intermediate language to realize data model functionality, and to implement part of the database management system kernel. Three main subjects have been covered: trigger definition languages, their execution models, and the optimization of trigger systems.

Horn logic is most suited for the specification of database conditions, while event definition would benefit from a temporal logic. The expressiveness of the trigger action language depends on the envisioned use. In many situations an extended query language suffices. However, direct database access is sometimes needed to permit control over the physical storage structures.

The choice of a suitable execution model for trigger systems also depends on the trigger application area. For, integrity enforcement triggers and storage optimization triggers pose widely differing constraints.

Our survey shows that to expose the different trigger uses and the imposed constraints on the execution model more research is required. In particular, lack of documented experience in applying triggers in real applications calls for more attention. A good starting point for such experiments is Hipac. Furthermore, a formalization is needed of the informally described execution models of the reviewed systems. The formulation of a formal description will clarify many issues and it will reveal inconsistencies. For example, what is the result of combining different TA schedules within one system?

To avoid opaque system behavior and to help the user with the development of trigger applications, design tools are urgently needed. To illustrate, one of the problems with trigger cascading is the possibility of cyclic execution, which may lead to a livelock. More theoretical research is needed to establish convergence criteria for cascading triggers and to develop a tool kit for livelock detection given a set of trigger definitions. Besides an in depth examination of the scope of execution strategies for multiple triggers, tools for checking their consistency should be developed as well.

The last subject of this survey is the optimization of trigger execution. The benefit of optimization partly depends on the kind of triggers defined in the system. Still, work remains to be done on the issue of efficient methods for implementing a trigger execution engine at the heart of a DBMS. A few research directions are : the optimization of the event history storage, reducing the search time for activated triggers by trigger indexing, and techniques to optimize the Rete algorithm. The latter two dimensions also indicate a major weakness in the systems being surveyed. Namely, they mostly seem to be designed as technical features without a thorough evaluation of the final result. No performance data on trigger systems has been published widely, nor a reference benchmark has yet been proposed.

Acknowledgements

We would like to thank Arno Siebes, Henri Bal and Irv Elshof for supplying feedback on earlier versions of this paper.

References

[Antunes, et al. 1990] F. Antunes, S. Baker, B. Cauldfield, M. Lopez, M. Sheppard. *A Pragmatic Approach for Integrating Data Management and Task Management: Modeling and Implementation Issues.* Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 1990, pp 427-436.

[Atkinson and Buneman 1989] M.P. Atkinson, O.P. Buneman. *Types and Persistence in Database Programming Languages.* ACM computing surveys, Vol. 19, No. 2, June 1987, pp. 105-190.

[Bancilhon and Ramakrishan 1986] F. Bancilhon, R. Ramakrishan. *An Amateurs Introduction to Recursive Query Processing Strategies.* Proceedings of the 1986 ACM SIGMOD international Conference on Management of Data, Washington, D.C., May 1986, pp. 16-52.

[Bertino and Musto 1988] E. Bertino, D.Musto. *Correctness of Semantic Integrity Checking in Database Management Systems.* Acta Informatica, Vol. 26, 1988, pp. 25-57.

[Cacace, et al. 1990] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. *Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm.* Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data, Atlantic City, NJ, May 1990, pp. 225-236

[Carey, et al. 1986] M.J. Carey, D. Frank, M. Muralkrisna, D.J. DeWitt, G. Graefe, J.E. Richardson, E.J. Shekita. *The Architecture of the EXODUS Extensible DBMS.* Readings in Database Sysems, ed. M. Stonebraker, Morgan-Kaufmann Publishers, San Mateo, California, 1988, pp. 488-502.

[Caseau 1989] Y. Caseau. *A Formal System for Producing Demons from Rules in an Object-Oriented Database.* Proceedings of the First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, December 1989, pp. 188-204.

[Chakravarthy and Minker 1986] U.S. Chakravarthy, J. Minker. *Multiple Query Processing in Deductive Databases using Query Graphs.* Proceedings of the 12th International Conference on Very Large Data Bases, Kyoto, Japan, 1986, pp. 384-391.

[Chakravarthy and Nesson 1990] S. Chakravarthy, S. Nesson. *Making an Object-Oriented BDMS Active: Design, Implementation, and Evaluation of a Prototype.* Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 1990, pp. 393-406.

[Chung 1984] K.L. Chung. *An Extended Taxis Compiler.* M.Sc. thesis, Department of Computer Science, University of Toronto, Toronto, January 1984.

[Cohen 1989] D. Cohen. *Compiling complex database transition triggers.* Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon, pp. 225-234.

[Dayal, et al. 1988a] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M.Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M.Livny, R. Jauhari. *The HIPAC Project: Combining Active Databases and Timing Constraints.* SIGMOD Record, Vol. 17, No 1, 1988, pp. 51-70.

[Dayal, et al. 1988b] U. Dayal, A. Buchmann, D. McCarthy. *Rules are Objects too: a Knowledge Model for an Active, Object-Oriented Database System.* Proceedings of the Second International Workshop on Object-Oriented Database Systems, Bad Munster am Stein-Ebernburg, FRG, September 1988, pp. 129-143.

[Forgy 1982] C.L. Forgy. Rete: A Fast Algorithm for the many Pattern/many Object Pattern Match Problem. Artificial Intelligence, Vol. 19, 1982, pp. 17-37.

[Frost 1986] R.A. Frost. *Formalizing the Notion of Semantic Integrity in Database and Knowledge Base Systems Work.* Proceedings of the 5th British National Conference on Databases, Canterbury, England, 1986, pp. 105-127.

[Gabbay and Guenthner 1983] D. Gabbay, F. Guenthner (eds.). *Handbook of philosophical logic, Vol. 1,2,3,4.* D. Reidel Publishing Company, Dordrecht, 1983.

[Graefe and DeWitt 1987] G. Graefe, D.J. DeWitt. *The EXODUS Optimizer Generator.* Proceedings of the 1987 ACM SIGMOD International Conference on the Management of Data, San Francisco, May 1987, pp. 161-173.

[Hanson 1989] E.N. Hanson. *An Initial Report on the Design of Ariel: a dbms with an Integrated Production Rule System.* SIGMOD Record, Vol. 18, No. 3, September 1989, pp. 12-19.

[Hanson, et al. 1990] E.N. Hanson, M. Chaabouni, C.H. Kim, and Y.W. Wang. *A Predicate Matching Algorithm for Database Rule Systems.* Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data, Atlantic City, NJ, pp. 271-280.

[Hudson and King 1986] S.E. Hudson, R. King. *Cactis: a Database System for Specifying Functionally-Defined Data.* Proceedings of the Workshop on Object-Oriented Database Systems, Pacific-Grove, California, September 1986, pp 26-37.

[Hudson and King 1989] S.E. Hudson, R. King. *Cactis: a Self-Adaptive, Concurrent Implementation of an Object-Oriented DBMS.* ACM transactions on database systems, Vol. 14, No. 3, September 1989, pp. 291-321.

[Hulsmann and Saake 1990] K. Hulsmann, G. Saake. *Representation of the Historical Information Necessary for Temporal Integrity Monitoring.* Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 1990, pp 378-392.

[Hsu and Imielinski 1985] A. Hsu, T. Imielinski. *Integrity Checking for Multiple Updates.* Proceedings of the 1985 ACM SIGMOD International Conference on the Management of Data, Austin, Texas, May 1985, pp. 152-168.

[Hsu, et al. 1988] M. Hsu, R. Ladin, D. McCarthy. *An Execution Model for Active Database Management Systems.* Proceedings of the 3rd International Conference on Data and Knowledge bases, June 1988.

[Ioannidis and Sellis 1989] Y.E. Ioannidis, T.K. Sellis. *Conflict Resolution of Rules Assigning Values to Virtual Attributes.* Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon, pp. 205-214.

[Jarke, et al. 1989] M. Jarke, M. Jeusfeld, T. Rose. *Software Process Modeling as a Strategy for KBMS Implementation.* The First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, December 1989, pp. 496-515.

[Kiernan, et al. 1990] G. Kiernan, C. de Maindreville, and E. Simon. *Making Deductive Database a Practical Technology: a Step Forward.* Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon, pp. 237-246.

[Kifer and Lozinskii 1989] M. Kifer, E.L. Lozinskii. *RI: A Logic for Reasoning with Inconsistency.* SIGMOD Record, Vol. 18, No. 3, September 1989, pp. 253-262.

[Kotz, et al. 1988] A.M. Kotz, K.R. Dittrich, and J.A. Mulle. *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism.* Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 1990, pp 76-91.

[Laguna Beach 1989] The Laguna Beach Participants. *Future Directions in DBMS Research.* SIGMOD Record, Vol. 18, No. 1, March 1989, pp. 17-26.

[Lipeck 1988] U.W. Lipeck. *Transformation of Dynamic Integrity Constraints into Transaction Specifications.* Proceedings of the 2nd International Conference on Database Theory, Bruges, Belgium, September 1988, pp. 322-337.

[Manola and Dayal 1986] F. Manola, U. Dayal. *PDM: an Object -riented Data Model.* Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific-Grove, California, September 1986, pp. 18-25.

[McCarthy and Dayal 1989] D.R. McCarthy, U. Dayal. *The Architecture of an Active Data Base Management System.* Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon, May 1989, pp. 215-224.

[Moss 1981] J.E.B. Moss. *Nested Transactions: an Approach to Reliable Distributed Computing.* PhD thesis 260, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[Mylopoulos, et al. 1980] J. Mylopoulos, P.A. Bernstein, H.K.T. Wong. *A Language Facility for Designing Database Intensive Applications.* ACM transactions on database systems, Vol. 5, No. 2, June 1980, pp. 185-207.

[Nicolas 1982] J.M. Nicolas. *Logic for Improving Integrity Checking in Relational Data Bases.* Acta Informatica Vol. 18, 1982, pp. 227-253.

[Nixon 1983] B.A. Nixon. *A Taxis Compiler.* M.Sc. thesis, Department of computer science, University of Toronto, Toronto, April 1983.

[Nixon, et al. 1987] B.A. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Hylopoulos, M. Stanley. *Implementation of a Compiler for a Semantic Datamodel: Experiences with Taxis.* Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, San Francisco, May 1987, pp. 118-131.

[Pu 1986] C. Pu. *Replication and Nested Transactions in the Eden Distributed System.* PhD thesis, University of Washington, DC, 1986.

[Risch 1989] T. Risch. *Monitoring Database Objects.* Proceedings of the 15th Conference on Very Large Data Bases, Amsterdam, Holland, August 1989, pp. 445-453.

[Rosenthal, et al. 1989] A. Rosenthal, U.S. Charkravarthy, B. Blaustein, J. Blakely. *Situation Monitoring for Active Databases.* Proceedings of the 15th Conference on Very Large Data Bases, Amsterdam, Holland, August 1989, pp 455-465.

[Segev and Gunadhi 1989] A. Segev, H Gunadhi. *Event-Join Optimization in Temporal Relational Databases.* Proceedings of the 15th Conference on Very Large Data Bases, Amsterdam, Holland, August 1989, pp, 205-216.

[Small 1986] C. Small. *An Implementation of a Constraint Enforcement System.* Proceedings of the 5th British National Conference on Databases, Leeds, 1986, pp. 141-154.

[Stemple, et al. 1987] D. Stemple, S. Mazumdar, and T. Sheard. *On the Modes and Meaning of Feedback to Transaction Designers.* Proceedings of the 1987 ACM SIGMOD International Conference on the Management of Data, San Francisco, May 1987, pp. 374-386.

[Stonebraker and Rowe 1986] M. Stonebraker, L. Rowe. *The Design of POSTGRES.* Proceedings of the 1986 ACM Sigmod Conference on Management of Data, Washington,D.C., May 1986, pp. 340-355.

[Stonebraker et al. 1988] M. Stonebraker, E. Hanson, C.H. Hong. *The design of the POSTGRES rules system.* Readings in Database Systems, ed. M. Stonebraker, Morgan-Kaufmann, San Mateo, California, pp. 556-565.

[Stonebraker et al. 1990] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. *On Rules, Procedures, Caching, and Views in Data Base Systems.* Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data, Atlantic City, NJ, May 1990, pp. 281-290.

[Sun, et al. 1989] X.H. Sun, N. Kamel, and L.M. Ni. *Solving Implication Problems in Database Applications.* Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Atlantic City, NJ, May 1990, pp. 185-192.

[Ullman 1980] J.D. Ullman. *Principles of Database Systems.* Computer science press, United States of America, 1980.

[Widom and Finkelstein 1990] J. Widom, S.J. Finkelstein. *Set-Oriented Production Rules in Relational Database Systems.* Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data, Atlantic City, NJ, May 1990, pp. 259-270.

[Zhou and Hsu 1990] Y. Zhou, M. Hsu. *A Theory for Rule Triggering Systems.* Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 1990, pp. 407-421.

[Zertuche and Buchmann 1990] D.R. Zertuche, A.D. Buchmann. *Execution Models for Active Database Systems: a Comparison.* GTE Laboratories, Waltham, January 1990.