Promises and Realities of Active Database Systems

Eric Simon
INRIA Rocquencourt
France
email: eric.simon@inria.fr

Angelika Kotz-Dittrich
Union Bank of Switzerland,
Switzerland
email: dittrich@ubilab.ubs.ch

Abstract

We confront the promises of active database systems with the result of their use by application developers. The main problems encountered are insufficient methodological support in analysis and design, the lack of standardization, missing development and administration tools for triggers, and weak performance. We concentrate on performance because we discovered it is one the main reasons that makes users reluctant to use active rules in the development of large applications. We show, using simple concrete examples, that optimizing large applications is rendered difficult by the separation of transactions and triggers and the misunderstanding of their subtle interactions. We argue that tools, which provide assistance to programmers, database administrators, and database designers to optimize their applications and master application evolution is strongly needed.

1 Introduction

The field of active database systems that originated in the mid-70 [Esw76] has for the last ten years received an increasing interest from both database vendors and database researchers. A large number of research projects are ongoing to design and implement relational or object-oriented active database systems (see [WCD95] for an overview). Many relational products already incorporate some limited form of active rule processing, and promote the active rule functionality as a key value of their system. Rules are also a prominent feature of the SQL3 standard [ISO94], currently under development. Finally, users have started using active rules in the development of real-life applications.

Active database systems have been presented as a very promising technology [Day88], [Sto92], and [WCD95]. They are expected to facilitate the design and maintenance of business rules, improve the reliability of applications regarding the enforcement of business rules, and enhance their performance. In this paper, we analyze the gap that exists between the potential benefits of active database systems and the actual capabilities of existing systems in the light of their use in the development of real-life applications. Our goal is to derive challenging research topics that we think should contribute to better establish the technology and encourage its dissemination.

We view an active database system as a black box and consider users that are either database designers, database administrators, or application programmers. Therefore, we study the problems that arise when users want to design a database schema including triggers, program transactions that automatically invoke triggers, verify the correctness of applications, optimize the performance of applications, and maintain applications, e.g., when transactions or triggers are changed. From this, we derive requirements for trigger languages, analysis and design methodologies, and development and administration tools.

Along this paper, we consider a relational active database framework and most of our observations take their roots in the study of application development projects in the banking environment. Consequently our concrete examples are mainly inspired from banking applications. However, we believe that most of the problems listed in this paper have their counterpart in object-oriented active database systems, and in other application domains

The paper is organized as follows. Section 2 states

Proceedings of the 21st VLDB Conference
Zürich, Switzerland, 1995

the expected benefits of active database systems, and describes several potential application domains. Section 3 presents the difficulties encountered by application developers using existing active database systems. Section 4 focuses on the need for administration tools that enable to optimize and maintain large active applications. Finally, section 5 concludes.

2 Promises of Active DBMS's

2.1 Passive vs Active Database Applications

Though most database management systems today have started to offer limited active functionality in the form of triggers, rules or similar, most database applications are still passive. By a *passive database application* we understand an application that does not make use of any active features even though the underlying DBMS may offer them. In contrast, an *active application* in our sense is not only based on a DBMS with active capabilities, but actually makes use of these capablities.

Passive DB applications use the DBMS only to create, retrieve, modify, and delete data by issuing corresponding operations. In particular, a considerable part of the *business rules*[1] essential to guarantee data quality and correct behaviour are embedded into the application programs. It can be observed that applications that ignore the availability of active features also tend not to make full use of other features like referential integrity.

There are two commonly used approaches followed in passive applications. The most frequent approach is to encode business rules using *database procedures* explicitly invoked from within a transaction. For instance, a procedure can be called before or after every modification to the database, or before committing the transaction. Appropriate actions (e.g., an abort of the transaction) will - again explicitly - be taken in response to the execution of a procedure that checks some condition.

A second approach is to periodically poll the database in order to check and apply business rules. For instance, companies send monthly retirement payments for their employees to a life insurance company. Sometimes the data sent are incomplete or incorrect. The strategy followed by some companies is to register all data into the database, and then run a separate application process that mines the database in order to discover anomalies which are subsequently handled either by dedicated "repairing" software (e.g., expert systems) or by humans. The rationale for this approach is to afford a high transaction throughput for a very large on-line database, given that the percentage of anomalies found in the database is reported to be below 1%. In contrast, incorporating controls in transactions would make transactions longer, thereby degrading the performance of applications. The problem with the "polling" approach is the difficulty to tune it: inconsistencies are introduced in the database and one has to carefuly control the consequences of that for all transactions.

It is worthwhile noticing that with passive database applications, programmers have the full control and responsibility of the application semantics, including the quality of stored data. Programmers also master how the processing of business rules is optimized within the application program.

In contrast, active database applications externalize part of their semantics and control structure delegating it to the database system. They rely on the definition and monitoring of *triggers*, also called *event-condition-action rules*, which are rules with an event that causes the rule to be triggered, a condition that is checked when the rule is triggered, and an action that is executed when the rule is triggered and its condition is true. When the event part is omitted, the rule is called a *condition-action* rule, whereas when the condition part is omitted, it is called an *event-action* rule. Typical actions are database modifications, procedures or a rollback statement that aborts the transaction. Events are issued by transactions and generally consist of database statements such as data modifications, data retrievals, or transactional commands. At specific points in a transaction's execution the database system takes a set of events issued by the transaction, automatically retrieves the triggered rules, and processes them. There are two kinds of rule processing points: rules can be triggered immediately after (or before) each occurrence of an event in the transaction (*immediate* execution mode), or at the end of the transaction (*deferred* execution mode). Triggers are defined as immediate or deferred and this determines subsequently their execution mode in a transaction. In most active database systems (and at least, in all commercial active database systems), the execution of triggers is done within the triggering transaction. We refer the interested reader to [WCD95] for a comprehensive view of triggers and active database systems.

2.2 Advantages of Active Database Systems

In this section, we discuss the benefits that active database systems can bring to applications. In the rest of the paper, we will elaborate on the many difficulties encountered with actual active database systems as well as that complex optimization problems still to be solved. However, in the opinion of the authors, the

[1]In this paper, the term *business rules* cover semantic integrity constraints as well as statements about how the business is performed.

advantages described below justify the efforts necessary to solve the subsequently mentioned design and optimization problems.

As a first benefit, *triggers enable a uniform and centralized description of the business rules* relevant to the information system. In fact, several conceptual modelling and information systems methodologies are being extended to handle restricted forms of triggers [TKL90, HKMS94]. Triggers rely on the use of query constructs for expressing condition and action parts of rules. Their regular format can be exploited to understand how rules relate to events, or how rules interact with each other, for instance by analyzing the relationships that a rule action has with the event or condition parts of other rules. This knowledge is useful for checking the correctness of rules, and tuning performance. In contrast, when business rules are embedded into application programs, they can be specified and implemented in a different way in several applications. It is therefore difficult to get the specification of rules validated by users, verify that they are consistently implemented, and optimize their global behaviour.

As a second benefit, *the use of triggers facilitates the maintenance of business rules.* Since triggers are modular, adding a business rule amounts to defining new triggers that will automatically be invoked by application programs when necessary. On the contrary, adding (changing, or removing) a constraint in a passive application requires to change application programs. Early studies have reported that a substantial maintenance effort in passive OLTP applications is spent in the maintenance of integrity controls.

For a further advantage, triggers are reliable since they are automatically invoked whenever an appropriate event is issued by a transaction. This provides a safe way to *ensure that every application obeys specific rules*, regardless of the method used to access the database. Declarative integrity constraints (also called assertions) also do this but are limited in what they can control. On the contrary, with passive applications, the correct enforcement of business rules is guaranteed only if every single transaction implements it correctly. This makes data quality dependent on the reliability of programmers and programming methodologies and may be the reason of severe inconsistencies as to the enforced policies.

Finally, triggers are expected to *improve the performance of applications.* There are two main reasons for that. The first reason is rooted in the *centralization of application semantics by means of triggers.* Due to centralization, more and better optimization techniques can be applied, redundancy of checking and repair operations can be avoided, and changes in the environment (like the fact that a checking operation is no longer necessary) can more easily be incorporated. Audits conducted on very large passive database applications have shown that transactions often perform more controls than necessary. In fact, the number of database procedures invoked from succeeding releases of these applications uses to increase monotonically. Calls to database procedures are rarely removed from transactions, though it turns out that changes in the data acquisition process have made some controls obsolete. Discovering such situations requires a lot of effort usually not considered as deserving in individual application programs.[2].

As a second argument, one *use of triggers is as an effective tuning instrument* to make the application run faster [Sha92]. A typical example is to replace a polling transaction that impedes the transactional traffic by triggers. Suppose to have an application that wants to display the latest data inserted into a table *SELLS* (insert_time, ...). A polling transaction would select data from *SELLS* since the last time it looked at the table. This transaction will conflict with inserters and create inter-transaction blockings. Furthermore, if polling is done too rarely, recently inserted records may be deleted by some transaction before they have been displayed. An alternative is to use a trigger that displays inserted data whenever an insert occurs to *SELLS*. The trigger avoids concurrency conflicts since it executes within the same transaction that inserts into *SELLS*.

Another familiar example [Sha92] is to create materialized views and maintain them with triggers. Suppose we have two relations *ORDER* (ordernum, itemnum, qty, vendor), and *ITEM* (itemnum, price), and we frequently ask "the total dollar amount on order from a particular vendor". This query can be very expensive on the above schema. An alternative is to create a relation $TOTAL-VENDOR$ (vendor, amount) where amount is the dollar value of goods on order to the vendor. Each update to *ORDER* causes an update to this redundant relation, which can easily be maintained with a trigger.

2.3 Application Domains for Active DBMS's

From our experiences which mainly come from the banking environment, we can report about a number of existing and potential applications for active DB technology. Typical implementations we currently know about use sets in the order of some hundred triggers (in three different banks, we came across applications using, e.g., 150, 200 and 220 triggers, respectively). Besides dedicated financial applications such as account management or the management of guarantees

[2]The hidden rule is often that it is preferable to pay extra processing cost for superfluous checks rather than endangering the correctness of data by missing any useful controls.

in international markets, these implementations also include more technical domains like the management of a bank's inhouse communication network.

We observed that the major usage of triggers in running applications is integrity constraint checking[3] and mostly referential integrity, alerting (i.e., rules whose action part only consist of messages), maintenance of statistical data and materialized views (mostly using event-action rules), as well as pre- and postprocessing of database updates.

Compared to the still small number of projects where triggers are already in use, we encountered a much larger number where active rules are now under consideration and first experiments have been started. In the sequel, we will examine a few of these potential application domains. Each time, we point out the benefits that can be expected from the use of an active DBMS. For more examples illustrating the application of active DBMS's in financial applications see [CS94].

Note that we decided to illustrate the applications by simple examples to give a flavor of the problems in principle. Be aware that the triggers encountered in practice are of a much higher degree of complexity than those examples and, even more essentially, that the large number of such triggers adds to the complexity.

The first application scenario (henceforth called "market watcher") is the electronic stock exchange or any financial trading environment with stock prices provided by a ticker service. The decision to buy or sell stocks must be based on the data representing the recent market trends. The traditional solution relies on human supervision, i.e., traders constantly watch data on the screen or poll the database by regularly submitting queries. More recent solutions try to automate this by installing processes that automatically poll the database. Clearly, triggers can be helpful to supervise the market trends by either notifying the human trader, or (to a certain extent) automatically kicking off the deals. We give a simple example of a trigger "watching" a *PRICES* relation. In our examples, rules have the form: on *event* if *condition* then *action*. We use a natural English language syntax for events, conditions, and actions in order to be independent from any system-specific trigger language. In a realistic environment, the number of triggers will depend on the number of financial instruments and the number of traders' strategies and can easily reach some ten thousands.

```
on insert to PRICES
if the price for Microsoft stocks is larger
   than the price for IBM stocks for the
   last 10 ticks
then notify trader A
```

[3] In one application this amounts to 77%.

The advantage of using triggers is that the conditions for trading decisions can be made explicit and can easily be inspected. An active DBMS can guarantee that interesting data constellations are never missed by the trader (provided that appropriate performance is guaranteed under real-time utilization).

A second example is portfolio management. Following a specific investment strategy (degree of risk, customer preferences, etc.) each portfolio is supervised and modified according to market opportunities. The investment strategy can be expressed (at least partly) as a constraint system on the minimum/optimum/maximum volume of different financial instruments in the portfolio (for example, the volume of bonds has to be between 10% and 15%, the volume of options less than 5%, the volume of gold preferably around 10%, etc.). In practice, such constraint systems tend to get rather large, involving an ever growing number of financial derivatives, foreign currencies, etc. Triggers can help to automatically supervise the constraint set, notify the portfolio manager when constraints are about to be violated, suggest modifications of the portfolio to approach the optimum, or prevent violating transactions. We give two examples of rules below. Again note that the number of triggers will grow rapidly with the number of investment stategies and the variety of new financial instruments and derivatives.

```
on update to PORTFOLIO.bonds
if bonds < 10% or bonds > 15%
then rollback

on update to PRICE.gold
if gold < threshold and
   PORTFOLIO.gold < optimum - 5%
then notify
```

In many financial applications, the notion of time plays an important role, either for timely reactions, or reactions based on historical data. We give below an example of triggers that handle time-related events and conditions over the database. In large financial institutes, there are a large number of deadlines and time limits to supervise, especially accumulating around specific points like end of business hours or end of month processing.

```
on end_of_month + 2 workdays
if balance is not available from branch Z
then notify

on update to CUSTOMER.address + 1 day
then send new forms to customer
```

As to historical data, time series analysis provides another attractive application domain for triggers. Time series on stock prices or macroeconomic data are analysed to produce forecasts and to base decisions on interesting historical developments. Triggers can be used to automatically notify the analyst on historical trends based for instance on moving averages, as shown below.

```
on insert to TIMESERIES
if moving average over last 30 days equals
   moving average over last 120 days and mo-
   ving average over last 30 days is rising
then suggest to sell
```

For this area, good performance is again important as the conditions are usually quite complex and extend over a possibly large time window in the historical database. The competitive advantage resulting from timely notification as provided by the triggers can be tremendous in today's highly volatile markets.

As a last application domain not particularly related to financial services but nevertheless of high relevance to financial institutions we would like to mention workflow management. The execution and monitoring of business processes in large enterprises is currently what you would call a "hot topic". An essential capability of workflow management systems is to kick off processes when certain events have happened (like the event that previous processes have terminated) and certain conditions are satisfied. With workflow specifications stored in an active database the control structure between processes can be guaranteed by triggers like the one shown below. An operational workflow system with hundreds of workflow specifications and intricate dependencies between processes will yield a large complex trigger set.

```
on PROCESS_A.terminate and PROCESS_B.terminate
if Process_A.sucessful and
   input_data_is_available
then start PROCESS_C
```

Numerous other applications can be found in environments like insurances (e.g., entry and administration of damage claims), healthcare (e.g., management of file's patients in an hospital), etc.

Though the potential benefits of specifying business rules using triggers seem obvious, developers keep asking us questions like "how should business rules be implemented?", "should we use the trigger mechanisms offered by database systems?". We have come across strategic guidelines in companies that categorically recommend not to code business rules into triggers at all though the reasons for that decision remain more or less fuzzy. Sometimes, in the same companies, designers are advised to describe business rules (e.g., on paper) using the concept of trigger. At this stage, our answer is to "code an application with triggers when the benefits mentioned earlier (uniform and centralized definition, maintenance, guaranteed invocation, effective tuning) are of importance". The crucial point here is to know whether the practical reality of active database systems match the expected benefits of triggers, which is the topic of the next section.

3 Realities of Active Database Systems

In this section, we examine the realities of active database systems in the light of their use in the development of applications. We decompose the problems found with active DBMSs into three categories. One is concerned with the design of active applications. The next one deals with the problems of security, reliability, and unpredictability. Finally, we address the performance problems.

3.1 Designing Applications

A first problem is the *lack of expressiveness of the trigger languages* provided by existing DBMS's. First, condition-action rules are usually not directly expressible. This problem is emphasized by restrictions of the trigger language, e.g., the event part of a rule must be associated with a single relation, or a disjunction of elementary events (even for the same relation) is not allowed. Coding a simple business rule, such as "if an employee earns more than his manager then notify", may entail the definition of many triggers because one trigger is needed for every data modification event capable of violating the constraint. The proliferation of rules renders more difficult the verification of their correctness. Consequently, most development guides recommend not to use triggers for coding integrity constraints that can be expressed by means of assertions in the data definition language. Some authors (e.g., [CW90]) have proposed to automate the generation of triggers from the specification of declarative assertions. E.g., for the portfolio management application mentioned earlier, the (semi-)automatic mapping from a constraint set specification to actual trigger definitions might be very useful. Some degree of automatic generation (e.g., for referential integrity) is already available in several commercial database design tools. In several applications, we found business rules that could not be implemented in the trigger language because of the restrictions imposed to the event part (e.g., no conjunction of events, no time-related events).

Furthermore, in some systems, for every relation, the possible number of associated triggers is limited. The number of rules which can be triggered by any specific event is also limited (in many cases to one). We found this is a severe restriction in applications managing history relations (e.g., a Withdraw relation) or central data (e.g, insurance claims), where a single change to a relation can trigger a large number of actions. In the SQL3 proposal, there are eighteen distinct types of triggers available for each relation, plus additional "update" triggers for the different columns of a relation. However, multiple triggers for the same event are possible and their order of execution can be specified using priorities.

Finally, many systems only offer immediate triggers. One reason for that is the uncertainty about the implementation of deferred triggers in distributed transactions (triggering point wrt prepare-to-commit?). However, immediate triggers are sometimes considered to be inadequate and keep people from using triggers.

A second problem is the *lack of a simple, clear and standardized semantics for trigger languages.* Trigger languages vary considerably in their syntax and semantics[4] which make applications developed with triggers not portable from one system to another. However, developers can expect the upcoming SQL3 standard to alleviate these problems. An important difference is the level of granularity (tuple-level or statement-level) of triggering. For languages that have both, conflicts may occur yielding an incorrect or non-deterministic behavior as shown in [Hor94]. Languages also differ from the restrictions (usually not clearly justified) placed on triggers. Although these differences strongly determine the behavior and the possible usage of triggers, there is no clear indication on which style of design is appropriate for some given rule semantics. Finally, in systems that support both triggers and integrity assertions, the exact execution behavior of both is not clearly defined.

Many developers we talked to, e.g. concerning the applications in Section 2.3, like the "market watcher" or the portfolio management system, are asking for design guidelines and reference applications to find out how to use triggers even in the simple form currently available from commercial relational systems.

Below, we mention the most frequently asked questions:

- What kind and amount of semantics has to be externalized into triggers as compared with semantics that has to stay in the application? E.g. should all the stock prices be polled within the application or should each check be encoded into a separate trigger?

- Which are the criteria for deciding when to choose stored procedures and when to choose triggers? E.g., even if the basic decision is to store the code for price checking in the database, this code could either be invoked explicitly or triggered automatically.

- What conditions have to be observed for the design of correct and terminating trigger sets? E.g., in the portfolio example, contradicting conditions like bonds $< 10\%$ and bonds $>= 10\%$ should not both trigger a rollback action.

[4]Even when they have a close syntax, their semantics can be quite different.

- Is there a classification of constraints that require different treatment? E.g., should simple integrity constraints like deriving an account balance be treated differently from complex business rules like reacting to specific customer patterns?

- Which criteria are there for the complexity of triggers? Should expressions in constraints and/or actions be limited, should a trigger refer to no more than one relation, should the action touch no other data than the triggering transaction etc.? E.g., with the market watcher example, should the action be limited to notification (no automatic buying and selling), should each trigger be limited to touch only one financial instrument?

- Is it more advantageous to design transactions and triggers in close connection or to develop them in isolation from each other? E.g., can the transactions modifying a portfolio be coded and/or modified independently of the investment constraints?

- Are there quantitative design rules like optimal size of a trigger set (absolute size, number of triggers per relation etc.)? E.g., is a set of triggers corresponding to 1000 financial instruments multiplied by 100 traders with individual strategies feasible?

In our view, a design and maintenance environment for active databses must support a methodology that allows for the initial design and subsequent modification of triggers in close connection with the database schema and the transactions. Furthermore, formal verification techniques, simulators and optional enforcement of complexity limitations have to be provided. These requirements will be backed up by the following section.

3.2 Security, Reliability, and Unpredictability

Real life experiences show that both project managers and senior developers are often reluctant to use active DBMS facilities because they consider triggers as insecure, unreliable and unpredictable. In this respect, their reaction is the same as with deductive rules in expert systems or knowledge base systems because they wonder how a set of individual, isolated rules will interact with each other and with application programs in concrete situations. With active rules, this suspicion is even greater because these rules "act on their own" and may directly affect the real world.

For mission-critical financial applications like the ones mentioned in Section 2.3 where triggers may automatically execute stock deals, influence the structure of large portfolios or rate customers as non-creditworthy, this attitude is well founded. The same is true for applications in plant control, patient care or aviation systems. Without guarantee (or at least very high probability) of correctness and predictable, unambiguous behavior, triggers will not be used in these fields.

There are less critical ways of using triggers, e.g., triggers which react just by notifying a human user. Nevertheless, the impression that less security is needed may be misleading. To argue about it, we consider a trade support system similar to the market watcher from Section 2.3 which has recently been introduced at a bank for the New York stock exchange. In the beginning, for about the first three months, traders resented the new system. After that, they got accustomed to it to such an extent that they now consider it a major problem if the software goes down for a single day. Traders are now reported to completely rely on the information delivered by the system and to no longer cross-check the automatically generated buy and sell suggestions. As a result, generating incorrect notifications will have the same desastrous effect as erroneous automatically triggered deals. It is therefore crucial for an active DBMS to offer all kinds of support to make triggers reliable and predictable.

A first impediment to this requirement is the difficulty to validate a large number of rules. As an example, think of the portfolio management application with the modification that the triggered actions do not only notify or rollback, but automatically adjust the portfolio to the various constraints. In this case, contradicting constraints will cause the triggers to bounce, e.g. one trigger's action will violate the condition of another trigger and vice versa, causing the restructuring to continue indefinitely.

We will now mention a number of concrete problems that have to be solved in order to support the design of more reliable and predictable active applications. First, most active database languages provide few or no facilities for imposing a structure on the rules in the database schema. Rules can be structured according to their triggering operations (e.g., all rules triggered by an insert to a particular relation are grouped together). But this will be undesirable when a set of rules with different events correspond to the same integrity constraint. For example, when an investment strategy is changed, all triggers defined to impose this strategy must be identified and updated, or when a financial instrument is no longer traded, all corresponding triggers have to be removed.

Second, existing active DBMSs do not provide rule analysis tools that enable to predict how rules will behave in realistic scenarios. For instance, the authors in [BCMP94] report that in most of their examples, the first set of rules produced by the design was indeed looping[5]. Some papers propose techniques to predict if a set of rules is guaranteed to terminate or to behave deterministically. As noted in [WCD95], these techniques still have deficiencies and several improvements are needed.

The definition of isolated triggers tends to rely on implicit assumptions about constraints that are observed in the application environment at the time of trigger design. E.g., the termination of triggers related to two financial instruments may rely on the fact that the two instruments are never traded at the same stock exchange. However, a change of such real-world assumptions may easily occur at some later point in time invalidating the original trigger design. Therefore, design and monitoring tools must support the explicit extraction of constraints from a trigger set, the addition of user-supplied constraints as well as the supervision of constraint modifications and violations during the whole lifetime of the triggers.

A further point revealed by our study of real applications is the difficulty of understanding the behaviour of transactions in presence of triggers. Adding a rule may alter the correctness of an existing transaction if the rule is triggered by the transaction and modifies the database in a way that is not expected by the rest of the transaction. Thus, analysis tools are also needed to understand how rules interact with transactions.

It cannot be expected that formal verification tools will guarantee a correct behavior of triggers in all cases. Therefore, further components will be needed in an active DBMS to support simulation and testing of triggers together with their triggering transactions. The tracing of triggered executions at run time can help to discover dysfunctions. For instance, in a trading system, the conditions under which deals have been executed must be logged to be investigated and cross-checked regularly. Accumulation of incorrect reactions can easily be imagined (compare this to recent cases where the ruin of a bank was brought about by the decision of one trader - though definitely not with the help of an active DBMS).

Two further methods to make active rules more secure and reliable in critical cases are generated triggers and explicit limitations. The idea to generate lower level trigger definitions from higher level specifications has been mentioned before. At this higher level, more comprehensive verifications are possible. The portfolio scenario is a typical example for this approach.

Limitations to what a trigger may execute or access should not be imposed by the DBMS but be individually definable for each trigger or subset of triggers,

[5] In current systems, the maximum number of cascading triggers is bound, thus infinite triggering does not actually occur even if there is a loop.

depending on the application. Limitations may relate to the maximal number of cascading triggering of rules, the database elements that may be accessed by the trigger, operations that may be included in the action part, etc. For example, one might demand that all triggers on modifications of sensitive relations are neither allowed to write these relations in their action part nor to perform operations that are capable of triggering other rules. Or a trigger on modifications of a customer account may in its action part modify data of the same customer and specific global balance data. In certain applications one even wants to impose the restriction that all actions are either rollback or notify (think of triggers that check for inconsistencies in an accounting system where all irregular transactions must either be prevented or checked by a human supervisor).

Last, regarding security, triggers in general need to be protected from unauthorized accesses. In fact, most systems having triggers offer the possibility to associate privileges with users to define, modify, or consult triggers (e.g., using a "grant" command). This can be problematic. First, programmers may need to see which rules can be triggered by the transaction they write, in particular with immediate rules whose action consists of changing the database. Now, the programmer may have the privileges for executing a trigger but not for reading it. Second, if a transaction (or a statement) is rolled back by a trigger, the associated error message must take into account the level of confidentiality associated with that trigger. As an example of the latter, think of an employee in a bank executing a transaction on a collegue's account. In this case, the bank's strategy is not to reveal the total assets of the fellow employee (as would be the case with usual customers). However, a trigger which checks the available total assets and rolls back the transaction in case a limit is violated may implicitly reveal this information. Much remains to be done in that area.

3.3 Performance

One of the main reasons that makes users reluctant to use triggers in the development of large applications is their anxiety about performance. This feeling is consolidated by recent experiences conducted with the development of applications that involve several hundreds of triggers on various DBMS platforms (e.g., application of account management for large commercial customers). When developers compare the performance of the same application coded with and without triggers (i.e., all the checks and reactions to updates are programmed linearly in the application programs), they observe that the trigger-based version runs two to four times slower. As a consequence, many consultants recommend not to use triggers intensively although they are convinced by the functionality.

This disquiet deserves some analysis. A natural question is to wonder if the immaturity of the implementations of triggers suffices to explain such a gap of performance. In fact, the overhead taken by the binding between events and rules, and the retrieval of rules remains quite small[6]. Another possible track of investigation is the lack of experience of developers in the programming of triggers.

With respect to performance, we have to distinguish between two kinds of trigger-based applications. The first category is generally obtained when only a few triggers are selectively added to an existing passive application. With available active database technology, such applications do not encounter any performance problems and run satisfactorily without sophisticated optimization techniques.

However, the relevant active applications now coming into existence are one or more orders of magnitude larger in terms of defined triggers, ranging from hundreds to thousands of triggers. Some examples and reasons for this fact have been given in Section 2.3. In these applications, triggers are used for all kinds of tasks like coding integrity constraints, alerters, business rules, time constraints etc. With applications of this kind which actually intend to exploit active databases to full degree, we have observed that performance problems represent a severe obstacle. In the following, we will try to reason what is behind this performance deficiency and what needs to be done about it. In fact, our thesis is the following:

> **Thesis** : the separation between transactions and triggers renders difficult the *global optimization* of the application

In practice, designers define triggers from application semantics specification, and programmers code transactions knowing that some properties over data are guaranteed. Thus, design phases are separate, and the levels of abstraction provided by the language used for transactions and triggers are different. This separation complicates the tuning of active applications, i.e., the activity of reconsidering the design of data structures, triggers, and transactions to make the application run more quickly. In particular, it is hard, and sometimes impossible, to reproduce optimizations that programmers used to do in passive applications. Finally, there is no design methodology that guides application developers in the design of efficient active applications.

[6]Note however that the only measurements available to us actually concern a small number of rules.

4 Optimizing Active Applications

Tuning is a well known difficult activity that requires to have a comprehensive understanding of the components of a DBMS [Sha92]. We argue that triggers further complicate the picture. In this section, we show that most difficulties for optimizing applications come from the misunderstanding of the interactions that exist between triggers and transactions. We review effective tuning techniques, show how to apply them in active applications, and explain the precautions that must be taken. From that, we derive various requirements, e.g., for administration tools.

4.1 Relaxing of Constraints

Suppose we have a purchase transaction that withdraws an amount X from a given bank account, and a business rule saying that "the balance of a bank account must never become negative". Suppose the rule is implemented by a trigger that checks the balance whenever an insert occurs in relation $WITHDRAW$. Every time the transaction inserts a tuple into $WITHDRAW$, relation $ACCOUNT$ is read by the trigger. Therefore, the purchase transaction conflicts with transactions that update relation $ACCOUNT$ periodically, which entails transaction blockings. A good optimization is to relax the constraint in a controlled way, e.g., for small withdrawals. This approach requires the computation of a function that gives the proportion of transactions run in relation to the amount withdrawn (X, in our example). Then, depending on the degree of consistency desired for the application, designers may decide to add an extra condition on the withdrawal amount to the condition of the trigger. In our example, the balance might be checked only for withdrawals above 30$. If the remainder represents 45% of the withdrawals, the optimization will certainly improve the transaction throughput of the application.

However, two precautions must be taken with this approach. First, changing the definition of the trigger may impact the correctness of existing transactions that rely on the strict satisfaction of the integrity constraint. Thus, the change to the trigger must be notified to programmers who can then check that no incorrect behaviour is introduced in the application. This is also true with passive applications but there programmers have the full control on the implementation of the constraint relaxation. Second, imagine that withdrawals originate from different transactions. For instance, a withdrawal is issued either by a purchase transaction (using a credit card), or by an automatic teller machine. Both transactions perform inserts to $WITHDRAW$ but their policies can be different regarding the above integrity constraint: violations can be accepted for purchases but not for ATM transactions. If the condition of the trigger defined for inserts to $WITHDRAW$ is changed then the effect will be effective for *all* transactions that perform inserts to $WITHDRAW$. In our example, this will prevent the constraint from being relaxed. Dirty solutions may circumvent the problem by duplicating the $WITHDRAW$ table but such a decision may have important secondary effects on the design of the transactions. Thus, it would be useful *to enable the specification of the context of invocation within the event part of triggers.* Note that this problem does not occur with passive applications because programmers directly control when and how checks are performed within transactions.

4.2 Optimizing a Relational Schema

Another tuning technique is to create redundant data in order to speed up the evaluation of queries that involve costly operations.

Creating redundant data can also improve the evaluation of trigger conditions. Suppose we have a relation SEC_PRICES (securityNo, stockExchange, date, price), and a trigger that implements a "London_better_than_NY_rule":

```
on insert to SEC_PRICES
if a new price from London is inserted and it is
higher than the average price for the same secu-
rity in NY for the past 10 days
then notify
```

The evaluation of the trigger's condition involves several costly operations: a join on securityNo between the set of inserted tuples and SEC_PRICES, a selection on stockExchange and an aggregate. Creating a new relation, say $NY\_AVERAGES$ (derived from SEC_PRICES), which contains the average prices from NY stock exchange over the last 10 days, facilitates the evaluation of the trigger when new prices for London are inserted. One must check that the new price is higher than the value in $NY\_AVERAGES$. This involves only a join between two relations one of which is rather small. Thus, a different trigger can be defined when London prices are inserted into SEC_PRICES.

Additional triggers are however needed to maintain $NY\_AVERAGES$ up-to-date when SEC_PRICES is updated. A primary effect of this maintenance is that transactions that do not need to check the London_better_than_NY_rule (e.g., those inserting prices from NY) now have to maintain the redundant relation. Thus, the value of the decision depends on the proportion of transactions that benefit from the optimization with respect to transactions that have to maintain the redundant relation.

However, understanding the implications of this decision is delicate in an active application. In fact, insertions to SEC_PRICES can be caused directly by transactions that update this relation but also by transactions that update another relation which triggers the execution of a trigger that inserts tuples into SEC_PRICES. Thus: *it is useful to know which transactions may directly or indirectly cause some changes to any relation.*

4.3 Select Lower Isolation Modes

Previous tuning techniques concern the rewriting of triggers or the redesign of the relational schema. We now look at techniques concerning the writing of transactions.

When transactions follow the strict two phase locking protocol, they run in total isolation (SQL isolation degree 3) [GR93]. This protocol implies that before reading or writing a database item, the transaction must acquire a lock on the item and hold it until a particular *lock point* after which no new lock will be acquired. The performance effect of this protocol is to create inter-transaction blockings and deadlocks. Most database systems offer the possibility to run transactions with a lower degree of isolation. For instance, if a transaction run in degree 2 then its read locks are released just after the read operation. This diminishes the waiting time for transactions that want to write the same database item. So, when consistency is not sacrificed, selecting a lower degree of isolation is an effective tuning technique [Sha92], [GR93].

We analyze the implications of this technique on active database transactions. Suppose we have two relations $WITHDRAW$ and $ACCOUNT$. A purchase transaction inserts a tuple into $WITHDRAW$ and then updates the balance of the corresponding bank account. This transaction can be run in isolation degree 2 since it does not issue any read operation.

Now, suppose we add an immediate trigger:

```
on insert to WITHDRAW
if ACCOUNT.balance is less than the amount of the
   withdrawal
then rollback
```

Suppose the balance for an account X is 1000 when two occurences of the purchase transactions, called T1 and T2, start to run concurrently with the following history:

```
T1 - insert 500 into WITHDRAW for account X
T1 - execute immediate trigger
T2 - insert 700 into WITHDRAW for account X
T1 - execute immediate trigger
T1 - update balance in ACCOUNT
T2 - update balance in ACCOUNT
```

At the end of the execution, the balance for account X is negative. The reason is that when T1 executes its trigger immediately after the insert, it reads account X (value is 1000) and then releases its lock on X because the transaction runs in degree 2. When T2 executes its trigger, X is not locked and can be read (its value is 1000), thus T2 continues its execution. When T1 updates the balance, the value becomes 300 and when T2 does its update, the value of balance is -200.

First, observe that if the trigger was declared as deferred, i.e., it executes at the end of the transaction, the problem would not exist. Second, suppose that the transaction only does an insert to $WITHDRAW$ and we define two immediate triggers:

```
trigger1: on insert to WITHDRAW
  then update to ACCOUNT.balance

trigger2: on update to ACCOUNT.balance,
  if ACCOUNT.balance is less than the amount of
     the update
  then rollback
```

This implementation is correct even if the transaction runs in degree 2. The lesson learned from this example is that: *selecting lower isolation modes for active database transactions requires to understand the invocation relationships that exist between the transaction and the triggers.* In particular, note that optimizations that turn out to be correct at some point may become incorrect if the set of triggers is changed.

This problem is already acknowledged by many development guides as a source of unreliability when referential integrity is enforced by means of triggers. It is necessary for the programmer writing the trigger procedure to explicitly lock (degree 3 isolation) the appropriate data for the duration of the transaction, and this must be done in triggers for all related tables.

Note that this problem does not occur with passive applications since adding a trigger requires to redefine the transaction.

4.4 Chopping Transactions

Transaction length has some effect on performance: the more locks a transaction requires the more it will have to wait, and the longer it executes the more it will cause other transactions to wait. Making transactions shorter may improve the performance of concurrent transactions when blocking situations occur and is thus an effective tuning technique [Sha92]. However, chopping a transaction into separate transactions must be done cautiously and requires to have a clear understanding of what are the possible concurrent transactions. Otherwise, inconsistencies can be introduced in the database. When all transaction programs are known in advance, it is possible to automatically chop transactions into smaller transactions without sacrificing isolation guarantees [SLSV95].

In an active application, chopping is more complicated. Suppose a purchase transaction T first updates the balance of some bank account and then inserts a tuple into *WITHDRAW*. If the only possible concurrent executions are instances of T then it is safe to chop the transaction into an update transaction, T_1, and an insert transaction, T_2, since the relations involved in each statement are distinct. Suppose the following deferred trigger is added later:

```
on update to ACCOUNT.balance
if the balance becomes negative or if the total
amount of the withdrawals this week exceeds 1000$
then rollback
```

This trigger will be executed within T_1. However, the chopping of T into T_1 and T_2 is no longer correct. Suppose the total amount of withdrawals for account X is 800 \$, and the balance is 400 \$ when two instances of the (chopped) purchase transactions (noted T_1, T_2, T_1', and T_2') execute concurrently with the following history:

```
T1  - decrement balance of 200$
T1  - execute trigger
T'1 - decrement balance of 100$
T'1 - execute trigger
T2  - insert 200$ into WITHDRAW
T'2 - insert 100$ into WITHDRAW
```

At the end of the execution, the balance for account X is positive but the total amount of withdrawals is 1100 \$, which exceeds the authorized threshold value. Thus: *chopping needs to take into account the rules that are capable of being triggered by the transaction.*

Suppose now that every instance of a chopped purchase transaction for an account X can either execute concurrently with other instances of purchase that concern accounts different from X, or with debit transactions that simply increment the balance of some bank account. The above inconsistency problem cannot occur any longer, and isolation is guaranteed. However, the problem is that if we execute the trigger in T_1 then its condition is evaluated on a state of *WITHDRAW* which does not take into account the new withdrawal (only visible after T_2 executes). Thus, chopping T violates the internal consistency of the transaction with regard to the condition of the trigger. Thus: ***assuring the correctness of a chopping requires understanding the data dependencies between the conditions of triggers and the statements of the transaction.***

4.5 Immediate Processing of Rules

Defining a trigger as immediate may be a good optimization technique if the trigger rolls back the transaction when a particular condition is violated. Immediate processing enables the trigger to execute as soon as possible in the transaction which avoids waiting until the end of the transaction if the transaction must be rolled back. However, defining triggers as immediate may introduce inconsistencies.

Suppose we have a transaction that does the following operations:

```
x = select balance from ACCOUNT where ...;
insert into WITHDRAW ...;
if x < 1000 then ...  else ....
```

Suppose trigger1 in Section 4.3 was defined as deferred and one changes its definition into an immediate trigger. This change clearly alters the correctness of the transaction because variable x may not be anymore up-to-date after executing the trigger that reacts on insert to *WITHDRAW*. Thus: immediate triggers may introduce side-effects into transactions that are difficult to control manually.

Another problem with immediate rules is that the programmer must be aware of which rules can be triggered and what their effect is. In some sense, this subverts the original idea that the programmer should concentrate on the logic of the transaction without worrying about business rules that have been externalized in the database schema. This yields security problems because some business rules need to be protected against unauthorized accesses. For instance, the writer of a transaction that inserts employees may not be allowed to see which (immediate) rules are triggered by updates to employee's salaries (e.g., bonus rules). This requires: ***a mechanism that enables programmers to see both authorized rules and rules needed to program correct transactions.***

Some systems (e.g., Oracle version 7), provide a restriction to what a trigger can change. E.g., a before-row trigger cannot change any values provided by the triggering statement, and an after-row trigger cannot change a new column value. Under these restrictions, the above "subvertion" problems cannot occur. However, this prevents from the definition of rules that repair constraint violations. Repairing actions must be handled either by the application program, or by the use of multiple triggers and extra "book-keeping" data.

5 Conclusions

We analyzed the problems encountered in practice with the development of active applications. We first showed that applications are difficult to design due to the lack of expressiveness and clear semantics of existing trigger languages, and the absence of design methodologies. Users call for extensions of existing methodologies that enables the specification of business rules and give at least guidelines regarding their implementation.

Another problem is the insufficient security, reliability, and predictability oftriggers. There is a clear need for analysis tools that enable to predict how rules will behave in realistic scenarios. The static rule analysis techniques already proposed by the research community represent a step forward but are clearly insufficient in practice. Simulation, testing, and debugging tools are also needed. Furthermore, the security problems have received very little attention until now.

The last problem is performance, which is a major component of understanding why the usage of triggers remains limited to "niche" areas. Thus, solving this problem is a major challenge for the database community. We analyzed the difficulties of the optimization of active applications. In fact, the separation between transactions and triggers leads to what we call the *Iceberg Problem* in active DB programming. From the database designer perspective, the visible part of the application is the DB schema including triggers, and the immersed part is transactions. From the programmer perspective, the situation is reversed. In order to tune triggers, the designer needs to know which transactions invoke which triggers, and in what proportion. Providing the designer with the code for transactions is certainly not appropriate. Similarly, information about triggers, in particular immediate triggers, is necessary to program efficient and correct transactions. Providing the programmer with the code for triggers is again not appropriate. There is a need for tuning tools that assist both designers and programmers in the building and maintenance of optimized active applications.

Finally, efficient implementations of triggers in DBMS's are needed. New research perspectives regarding this problem are discussed in [LS95].

Acknowledgements: We wish to thank Mokrane Bouzeghoub, Eric Dujardin, Francoise Fabret, Irene Kunz, Francois Llirbat, and Dimitri Tombroff for the many constructive discussions we had on the topic of this paper. We also want to thank Phil Bernstein for his helpful pointer towards French banking organizations.

References

[BCMP94] E. Baralis, S. Ceri, G. Monteleone, and S. Paraboschi. An Intelligent Database System Application: the Design of EMS. In T. Risch and W. Litwin, editors, *Applications of Databases*. LNCS, Springer-Verlag, 1994.

[CS94] R. Chandra and A. Seguev. Active Databases for Financial Applications. In *Proc. of the 4th International Workshop on Research Issues in Data Engineering*, Houston, Texas, Feb. 1994.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proc. of the 6th Int. Conf. on VLDB*, Brisbane, Australia, Aug. 1990.

[Day88] U. Dayal. Active Database Management Systems. In *Proc. of the 3th Int. Conf. on Data and Knowledge Bases*, p. 150–169, Jerusalem, Israel, Jun 1988.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-running Activities with Triggers and Transactions. In *Proc. of the ACM SIGMOD Int. Conf.*, p. 204–214, Atlantic City, New Jersey, May 1990.

[Esw76] K.P. Eswaran. Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System. IBM Research Report RJ 1820, San Jose, California, Aug. 1976.

[GR93] J. Gray and A. Reuter. *Transaction Processing.* Morgan Kaufman, 1993.

[HKMS94] H, Herbst and G. Knolmayer and T. Myrach and M. Schlesinger. The Specification of Business Rules: A Comparison of Selected Methodologies. In A. Verrijn-Stuart and T. Olle, editors *Methods and Associated Tools for the Information System Life Cycle*, Amsterdam, 1994.

[Hor94] B. Horowitz. Intermediate States as a Source of non-Deterministic Behavior in Triggers. In *Proc. of Int. IEEE RIDE Workshop*, Houston, Feb. 1994.

[ISO94] ISO-ANSI working draft: Database Language SQL3, 1994. X3H2/94/080; SOU/003.

[LS95] F. Llirbat and E. Simon. Optimizing Active Database Transactions: A New Perspective. In *Proc. of Int. Workshop on Active and Real-Time Database Systems*, Skovde, Sweden, June 1995.

[Sha92] D. Shasha. *Database Tuning: a Principled Approach.* Prentice Hall, 1992.

[SLSV95] D. Shasha and F. Llirbat and E. Simon and P. Valduriez. Transaction Chopping: Algorithms and Performances Studies. In *ACM Transactions on Database Systems*, 20(3), Sep 95.

[Sto92] M. Stonebraker. The Integration of Rule Systems and Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):415–423, Oct 1992.

[TKL90] A. Tsalgatidou and V. Karakostas and P. Loucopoulos. Rule-based requirements specification and validation. In *Proc. of the 2nd Nordic Conf. on Advanced Information Systems Engineering*, Springer Verlag, LNCS N. 436, May 1990.

[WCD95] J. Widom and S. Ceri. *Active Database Systems.* Morgan-Kaufmann, San Francisco, 1995.