Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS

Ulf Schreier*
schreier@forwiss.uni-erlangen.de

Hamid Pirahesh
pirahesh@ibm.com

Rakesh Agrawal
ragrawal@ibm.com

C. Mohan
mohan@ibm.com

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120, USA

Abstract

Alert is an extension architecture designed for transforming a passive SQL DBMS into an active DBMS. The salient features of the design of Alert are reusing, to the extent possible, the passive DBMS technology, and making minimal changes to the language and implementation of the passive DBMS. Alert provides a layered architecture that allows the semantics of a variety of production rule languages to be supported on top. Rules may be specified on user-defined as well as built-in operations. Both synchronous and asynchronous event monitoring are possible. This paper presents the design of Alert and its implementation in the Starburst extensible DBMS.

1 Introduction

Passive database management systems (DBMSs) are *program-driven* — users query the current state of database and retrieve the information *currently* available in the database. Active DBMSs, on the other hand, are *data-driven* — users specify to the DBMS the information they need. If the information of interest is currently available, the DBMS immediately provides it to the relevant users; otherwise, the DBMS actively monitors the arrival of the desired information and provides it to the interested users as it becomes available. In other words, the scope of a query in a passive DBMS is limited to the past and present data, whereas the scope of a query in an active DBMS additionally includes *future* data. An active DBMS reverses the control flow between applications and the DBMS — instead of only applications calling the DBMS, the DBMS may also call applications in an active DBMS.

Active DBMS have recently been the focus of much research [CBB+89, HLM88, MD89, RCBB89, DBB+88, SLR88, SHP88, SHP89, SJGP90, WF90, WCL91, ZB90]. The research seems to be aimed at developing new language constructs, defining new execution paradigms, and devising new implementation techniques. Later, we survey some of the important results that have emerged.

Alert is an extension architecture, implemented in the Starburst extensible DBMS [HCL+90] at the IBM Almaden Research Center, for experimentation with active databases. Rather than starting from scratch, Alert takes an evolutionary viewpoint and extends a passive DBMS into an active DBMS. Alert takes advantage of the passive DBMS services to the extent possible and adds active elements to them only if necessary. For example, rather than introducing a new rule language [WF90, SJGP90], Alert introduces active queries to express triggers and unifies them with the regular passive queries so that active queries may be expressed in SQL with minimal additions to the language; rather than adding AI techniques like RETE network [For79] or designing entirely new techniques such as the tuple marking algorithm in [SHP88] for event detection, Alert employs indexing and query optimization techniques for the same purpose. Alert recognizes that the next generation DBMSs will have many features of object-oriented systems. In particular, the typed database objects may only be manipulated through the methods defined for them [ADL91]. Therefore, rather than limiting the event monitoring to low-level database operations like SQL INSERT, DELETE, and UPDATE on base tables as in [Syb87, ISO90, SJGP90, WF90], Alert allows monitoring of user-defined operations like *hire* specified on abstract objects like views.

Alert proposes a layered architecture as shown in Figure Figure 1. This architecture has been motivated by the need for providing efficient support to multiple rule languages and sharing the same database between them and non-rule-based applications. Given the commercial availability and success of several production languages such as KRL [BW77], OPS5 [For81], and KEE [IBM88], it is undesirable to require all users to use one rule-processing model. These languages have different semantics — they have different strategies for conflict resolution, modularization of rule sets, and execution modes.

The layered architecture proposed by Alert has a lower layer, the *monitor*, that provides the basic ser-

*Current address: AHP Havermann & Partner GmbH, Robert-Koch-Str. 1a, 8033 Planegg, Germany

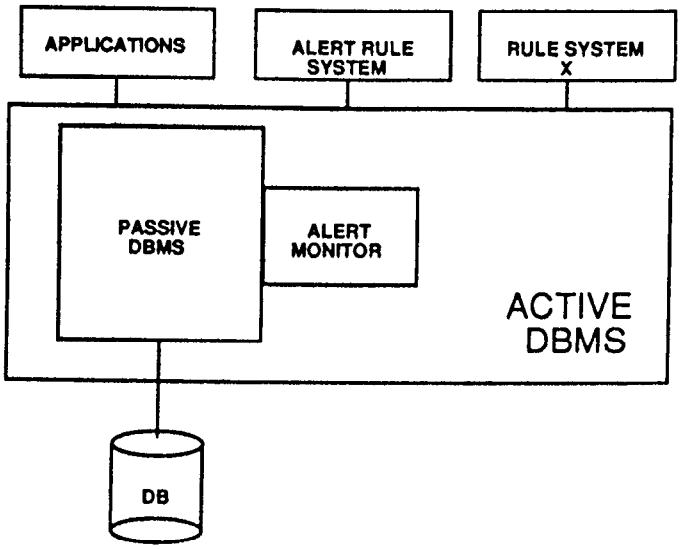

Figure 1: Architecture

vices. In particular, it provides different communication protocols between triggering and triggered actions: synchronous or asynchronous cooperation, deferred or immediate notification, and execution of the triggered action in the same or in a different transaction as the triggering action. Various rule languages form the upper layer and use the services of the monitor. The desired execution mode is communicated to the monitor at the time of rule activation. Language specific operations such as conflict resolution are not the responsibility of the monitor — they are carried out in the corresponding language system. This architecture is in contrast to an architecture such as that of KEE [IBM88], where the data from the database is extracted and cached in the rules system, and the cache is not shared between the rule system and the other users of the database.

The rest of the paper is organized as follows. We present Alert's extensions to the relational model in Section 2. Section 3 presents Alert's rule language, which is a slight extension of SQL with the primitives introduced in Section 2. Several examples of trigger definitions are included in this section. This section also presents the implementation of the Alert rule system in the Starburst extensible relational DBMS [HCL+90]. The Alert monitor component is presented in Section 4. Section 5 discusses related work. We conclude with a summary and directions for future work in Section 6.

2 Active And Passive Queries

Current relational DBMSs support only *passive* tables and *passive* queries. In standard SQL [ISO90], one can define a *cursor* for a passive query specified over one or more passive tables. Once a cursor has been opened, successive *fetch* operations yield tuples satisfying the query. Following the return of the first EOF (end-of-file) after all the tuples have been returned, all future fetches on this cursor yield EOF.

We introduce the notions of *active tables* and *active queries*. Active tables are append-only tables in which the tuples are never updated in-place and new tuples are added at the end. Active queries are queries that range over active tables. Active queries differ from passive queries in their cursor behavior. When a cursor is opened for an active query involving one or more active tables, tuples added to an active table after the cursor was opened also contribute to answer tuples. Thus, it is possible that a cursor opened for an active query gets an EOF in response to a fetch, but then gets new tuples in response to future fetches if new tuples are added to the underlying active tables. A fetch on a cursor opened for a passive query never returns tuples added to a table over which the query ranges, after an EOF has been returned. Thus, the active queries are defined over past, present, and future data, whereas the domain of the passive queries is limited to past and present data.

The standard SQL fetch can be viewed as a *non-blocking read*: if no more tuples are available in the answer set of the query, the process doing a fetch is not blocked but is simply returned an EOF. We introduce a new SQL primitive — *fetch-wait* — to iterate over active queries. The fetch-wait corresponds to a *blocking read*. The process doing a fetch-wait is returned a tuple if one is available. However, if the current answer set is exhausted, the process doing a fetch-wait is blocked until one becomes available. A fetch-wait returns EOF only if it is guaranteed that no more answer tuples will ever be generated.

Thus, by defining a suitable query over an appropriate active table, the user may monitor events being logged in the active table by opening the active query and applying the *fetch-wait* operation on the resulting cursor. The user may also monitor events in a polling mode by applying *fetch* operations on an opened cursor for an active query.

An active database contains user-defined active tables. The user-defined operations are logged in these tables. As with other database tables, the format of these tables is the user's responsibility, and applications are responsible for adding tuples to these tables. These tables are akin to journals created by many applications, such as banking transactions [TPC89].

In addition to the user-defined active tables, the DBMS creates an active table for every user-defined passive table. The system automatically logs all the system-defined operations on a passive table in the corresponding active table. These tables are accessible to the users, and the owner of a passive table is responsible for the management of the associated active table. In particular, the user may truncate old tuples to limit the size of an active table.

It is possible to define an active query that involves both active and passive tables. Some of the active tables in an active query may be system-defined and others may be user-defined active tables.

No new SQL syntax has been introduced to define active queries — syntactically they appear identical to passive queries. The only difference is that an active query includes at least one active table in its range. This unification of the language for active and passive queries is similar to the unification of forward chaining and backward chaining languages. We will further discuss this point as part of the discussion on the related research in Section 5. Due to this unification, active queries inherit the closure property of standard SQL, allowing active queries to be defined in terms of other active and passive queries.

We now give an example to illustrate active queries. Let there be an active table, *journal*, defined as follows:

```
Create Active Table journal
    ( ename string, company string, event string,
      purpose string, expense_amount integer,
      method_name string, timestamp integer,
      transid integer, date integer);
```

Assume a user-defined operation, *expense_claim*, defined as

```
expense_claim (ename, expense-amount, purpose, date):
```

This operation increments the number of trips taken by an employee, decreases the remaining travel budget of employee's department by the expense-amount, and appends a summary tuple to *journal*.

We can now define an active query, embedded in an application program, that retrieves a tuple whenever an expense_claim exceeds $1000 as follows:

```
declare C1 cursor for
   SELECT ename, purpose FROM journal
   WHERE method_name='expense_claim'
         AND expense_amount > 1000
open C1;

while (TRUE)
   {fetch_wait C1 into :ename, :purpose;
    /*This call completes when there is an answer
    /*do whatever is desired with the fetched tuple.*/
    ...
   }
```

This query first retrieves all the tuples in *journal* satisfying the given predicate, and then remains open to receive future appends to *journal* by the method *expense_claim*. Retrieval of the past data can be avoided, if desired, by using the timestamp column of *journal* in a predicate in the **Where** clause.

Observe that the definition and manipulation of an active query is identical to the passive standard SQL query, the only differences being that the active query has been defined over an active table and the **fetch_wait** call is used rather than **fetch**.

3 Alert Rule System

We have implemented a rule system, henceforth referred to as the Alert Rule System, based on the concept of active queries introduced in Section 2. This section describes how Alert rules are defined, activated, and executed.

3.1 Rule Definition

Alert rules are production rules that are fired when their conditions are satisfied. Syntactically, Alert rules are *named active queries*. They are defined by specifying rule conditions in the SQL **FROM** and **WHERE** clauses, and by specifying rule actions in the **SELECT** clause. The SQL view mechanism is used for naming the rules.

We give some examples of rules written in SQL enhanced with user-defined functions [HFLP89, ISO90]. We first define a rule **cost_watch** that sends mail to Irv, using the function **sbmail** whenever an expense claim for more than $1000 is filed. This rule is specified as an active query over *journal*. The rule condition is specified in the **WHERE** clause. The rule action of invoking **sbmail** is specified in the **Select** clause. This action is performed for each tuple selected by the query. The rule name is specified in the **Create rule** clause, that mimicks the **Create view** clause. Here is the rule definition:

```
Create rule cost_watch as
    SELECT sbmail('Irv',ename)
    FROM journal
    WHERE method_name='expense_claim'
      AND expense_amount > 1000
```

A rule, as defined above, *is* a SQL view. Hence, like views, it can be referred to in any other query. This definition **unifies** rules and views.

Since a rule is a query, it can refer not only to base tables, but also to *views* in its definition. It is an important feature, given the central role played by views in complex database applications. To illustrate a rule definition involving views, let us create a view **trip_events** on **journal**, which provides the public relations department information about all trips, but hides from them the expense amounts:

```
Create view trip_events (name, purpose, date, company, event)
    as
    SELECT ename, purpose, date, company
    FROM journal
    WHERE method_name='expense_claim'
```

Let us now define a rule on the view **trip_events**. This rule informs the public relations department about all the trips taken by Irv.

```
Create rule pr_watch as
    SELECT sbmail('prdept',name, purpose)
    FROM trip_events
    WHERE name = 'Irv'
```

We now give an example that illustrates the **closure** property and the rule reuse facility of Alert rules by re-using the definition of a condition in two different rules. Suppose Laura has defined a condition, called **laura_condition**, for checking whether an expense claim exceeds $2000. By defining this named condition separately as

```
Create rule laura_condition as
   SELECT ename, expense_amount, purpose
   FROM journal
   WHERE method_name='expense_claim'
         AND expense_amount > 2000
```

Laura can allow others to use the condition without their knowing exactly what that condition is. Laura

can use this condition in a rule, called **laura_watch**, as follows:

```
Create rule laura_watch as
   SELECT sbmail('Laura', ename, purpose)
   FROM laura_condition
```

So can Irv, who refines it with a stricter condition in his rule, called **irv_watch**:

```
create rule irv_watch as
   SELECT sbmail('irv',ename, purpose)
   FROM laura_condition
   WHERE expense_amount > 5000
```

Both references to **laura_condition** use standard SQL view nesting.

Since Alert rules output data as well as executing rule actions, they can be **nested** into quite complex rules. For example, we can define a rule **watch_hierarchy** that causes different levels of management to be informed, depending on the amount of travel expenses involved. If the expense amount is more than $2000 Laura is informed, and if the amount is more than $5000 Irv is also informed:

```
Create rule watch_hierarchy as
   SELECT sbmail('irv',ename, purpose)
   FROM irv (ename, expense_amount, purpose) AS
            (SELECT ename, expense_amount, purpose,
            sbmail('laura',ename, purpose)
            FROM journal laura
            WHERE laura.method_name = 'expense_claim'
            AND laura.expense_amount > 2000
            )
   WHERE irv.expense_amount > 5000
```

The conditions of Alert rules can even be a union of multiple events on multiple tables, as illustrated by the following **major_events** rule. It notifies the lab director if an expense_claim with amount more than $10000 is filed or if a new department is created. To do this, it makes use of the system-generated active table for logging the actions on the passive table **dept**. These system-defined tables are accessed using the notation **elog**(*table, operation*).

```
Create rule major_events as
   SELECT sbmail('Matisoo', 'Significant expense Alert',
                 name, amount)
   FROM dt(name, amount) AS
          ( SELECT ename, expense_amount
            FROM journal
            WHERE method_name='expense_claim'
                  AND expense-amount > 10000
          Union
            SELECT deptname, budget
            FROM et AS elog(dept,ins)
          )
```

The conditions of Alert rules can also involve joins of multiple active tables, as illustrated in the following **speech_watch** rule. This rule notifies a marketing director about the effect of a major speech given by an employee of a company on the stock of the company. We assume we have another active table containing stock transactions.

```
Create rule speech_watch as
   SELECT sbmail('Marketing Director', name,
                 stocktype,stockprice)
   FROM trip_events, stock
   WHERE trip_events.event='Major speech'
         AND trip_events.company=stock.stocktype
         AND within_week(trip_events.date,stock.date)
```

This rule returns an employee's name who has given a major speech, and information about the stocks traded within a week of that speech. Observe that both operands of the join are active tables in this rule. When a major speech tuple is inserted, the rule joins it with the stock transaction tuples for the week before, and when a stock transaction is inserted, the rule joins it with the major speech tuples for the week before. The rule sends a notification if there is a match in either case.

In all the examples considered above, the rule action was specified to be a scalar function [HFLP89, ISO90]. However, the action can be any SQL DML statement. The action can also be any general program with embedded active SQL queries in them, including table functions [HFLP89].

3.2 Rule Activation

Before a rule can fire, it must be activated. We have added two new SQL commands to activate and deactivate rules. The **Activate** command takes a named active query, opens it, and sets up an iteration in the *fetch-wait* mode. It also returns a handle that can be used to deactivate the rule. The **Deactivate** command uses the handle to close the query. It is possible to create several simultaneous activations of a rule since a SQL query may be opened several times.

With each activation of a rule, it is possible to specify the attributes of the execution. The user may define the transaction coupling mode, the time coupling mode, and the assertion mode for the rule execution, if and when the rule is fired. A rich set of rule execution attributes were identified in HiPAC [CBB+89]. However, in HiPAC, the execution attributes are **rule** properties, whereas they are **activation** properties in *Alert*. Thus, the same rule may be activated differently by different users.

The details of the execution modes will be discussed momentarily, but let us first give the syntax for **Activate** and **Deactivate**:

```
Activate
   <RuleName>,
   Transcoupling = Same | Separate,
   Timecoupling = Sync | Async
   Assertion = Immediate | Deferred
```

The **Activate** command returns a unique *rule_id* for each rule activation. This *rule_id* is used in the **Deactivate** command:

```
DEACTIVATE    <rule_id>
```

The defaults for transaction coupling, time coupling, and assertion mode are Same, Synchronous, and Immediate respectively. We now discuss the details of these execution attributes:

- **Transaction Coupling Mode**: *Separate, Same.*

  A rule is triggered due to state changes made to a database by some *triggering transaction* that causes the rule conditions to become true. From a DBMS viewpoint, the triggering and triggered actions are two applications that may be part of the same or different transactions. The transaction coupling mode specifies whether the triggered action should be executed as part of the triggering transaction, or as a separate transaction.

  The transaction coupling mode *Same* means that the triggered action runs as part of the triggering transaction. For instance, a rule that checks the limits on the travel expense items should run as part of transaction of expense_claim method, allowing it to correct any mistakes before it commits.

  Separate means that the triggered action runs as a separate transaction. Suppose there is an active table containing stock transactions. Stock brokers may define rules with conditions on price, stock type, etc, the actions being selling or buying stocks. Potentially, a very large number of such rules may be defined. A stock transaction may trigger one or more of these rules. The triggered rules should perform their actions in a separate transaction.

- **Time Coupling**: *asynchronous, synchronous.*

  The *synchronous* time coupling means that if a rule is triggered due to an action of a triggering transaction, the action of the triggered rule is executed, and the execution control returns to the triggering transaction only after the completion of the triggered action. For example, it is desirable to run synchronously a rule that checks the limits on spending items in order to detect any errors immediately.

  The *asynchronous* coupling means that the triggering action and the action of any triggered transaction runs in parallel. In the stock example above, the broker rules should run asynchronously.

- **Assertion Mode**: *Immediate, Deferred.*

  The assertion mode *immediate* means that the rule is triggered as soon as the rule condition is satisfied. This mode may be used, for example, to make sure that the raise given to an employee is not negative.

  The *deferred* assertion mode is used in conjunction with the same transaction coupling mode. This mode means that the rule condition is evaluated only in a region of a triggering transaction bracketed by *Begin-assert* and *End-assert* commands. The *Begin-assert* and *End-assert* commands are similar to the *Set constraint on* and *Set constraint off* commands in standard SQL. The deferred mode can be used for enforcing deferred constraints using rules [Mar90].

The same transaction coupling can be used only in conjunction with synchronous time coupling. Separate transaction coupling can be used only in conjunction with asynchronous time coupling. The deferred mode is applicable only in conjunction with the same transaction coupling mode. However, the immediate mode can be used in conjunction with both same and separate transaction couplings.

A much richer set of coupling modes have been proposed by the HiPAC project [CBB+89]. We feel that some of the HiPAC's coupling modes are not useful enough to justify their implementation complexity (condition testing in a separate transaction, for example), and some can be simulated using the coupling modes proposed here. For example, *nested top transaction* in HiPAC is equivalent to separate but synchronous coupling, a combination not directly supported by Alert . To simulate this coupling, we define a rule with *same and synchronous* coupling, and the first action of this rule creates a new top-level transaction.

3.3 Rule Execution

We now describe the implementation of how Alert rule system triggers and executes rules. The previous section explained how one defines and activates Alert rules. Alert Rule system stores in a catalog the definitions and activations. The catalog also has some state information - for example, if a deferred rule is in the asserting state or not.

Let us trace the control flow of a rule execution. First, assume that the rule has been specified with the attributes asynchronous, separate, and immediate modes. The active query associated with this rule is over the active table *stock* with predicate *stocktype='IBM'* **AND** *value>$150*. Assume that the stock table is initially empty. Upon the activation of the rule, the Alert rule system *opens* the active query associated with this rule. At some point, a triggering ticker-tape transaction inserts a tuple in the *stock* table with values *stocktype='IBM'* and *price=$151* (message (1) in Figure 2). As a result of this update to the database, Alert monitor checks the condition of all active queries on this active table to determine which ones have new data to process. In our example, the query associated with our rule qualifies. In general, there might be a set of such active queries. From the viewpoint of the Alert rule system, all the rules in this set can be triggered in some order. Alert monitor gives the list of qualifying active queries to the conflict resolver component of the Alert rule system, which determines the order of execution (Figure 2, message (2)). The conflict resolver saves this list and it returns the control immediately to Alert monitor, which in turn returns the control to the triggering action (messages (3) in Figure 2). The triggering action's transaction may now commit. Note that the triggering transaction does *not wait* for initiation of the triggered rule, making this execution asynchronous.

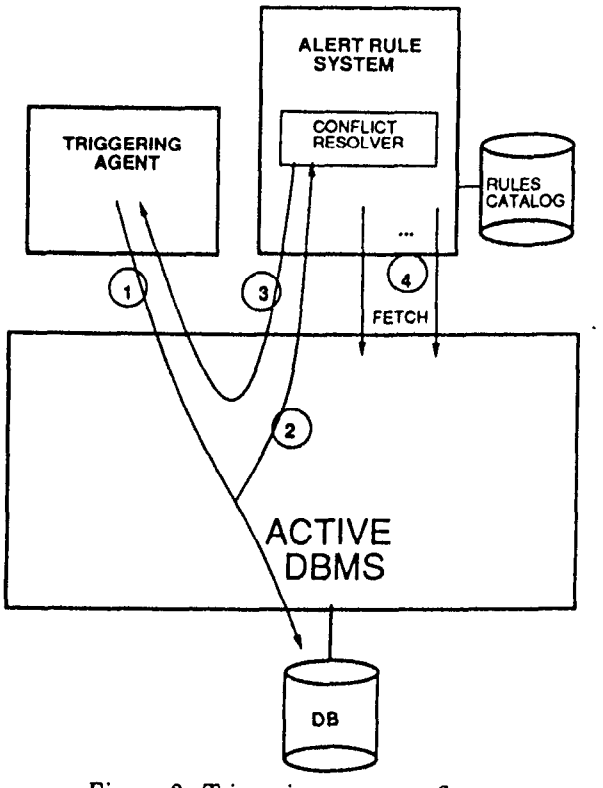

Figure 2: Triggering message flow

The conflict resolver chooses a rule to be triggered. Alert rule system does a *fetch* on that rule (messages (4) in Figure 2), causing its qualified tuples to be passed to the associated action. The processing of this rule stops when there are no more qualified tuples, resulting in an EOF to be returned to Alert rule system. Now, the conflict resolver can choose the next rule to be processed. This process of choosing and processing rules continues until no more rules need to be triggered, in which case the conflict resolver goes to sleep, waiting for another message from Alert monitor.

Now, let us examine the control flow for the *synchronous, same, immediate* case. The triggered rule must run as part of the triggering transaction. Therefore, the Alert rule system should open a new active query for each triggering transaction. But how can the Alert monitor inform the Alert rule system about this triggering event caused by a triggering transaction, if no active query has yet been opened for this rule? To solve this problem, Alert rule system does an extra open for the associated active query at the rule activation time, if the activation mode is synchronous same. No transaction is assigned to this open (i.e., it runs with degree 0 consistency [GLPT76]). Now, when a triggering transaction changes the database state, the Alert monitor sends a message to the the conflict resolver, which chooses a synchronous rule to execute. The rest of the control flow is similar to the *asynchronous, separate, immediate* case discussed earlier.

For rules activated with the *deferred* option, Alert rule system defers execution until the rules are in the asserting mode (i.e., between *begin-assert* and *end-assert* commands.). Otherwise, the behavior is the same as for *immediate*.

4 Alert Monitor Subsystem

We now describe the implementation of the Alert monitor. The Alert monitor has been implemented as an extension to the Starburst DBMS (see Figure 1). The Alert monitor manages the active tables associated with passive tables, and monitors changes to the database state.

The management of active tables is identical to the management of passive tables, except for insertions. For active tables, inserts should be done in such a way that the cursors on these tables see the inserted data. Active tables are implemented as table queues [PMC+90].

An active table may become the major source of lock contention. To see why, let us go through an example. Suppose a stock monitoring rule referencing table *stock* has a predicate: *stocktype='IBM'*, and has been activated to run as a separate transaction. A large number of tuples that will be appended to *stock* will not be of interest to this rule. When DBMS reads a tuple, a standard locking protocol would require that a lock be obtained on behalf of the transaction processing this rule. After the lock is granted, the associated page is latched for assuring physical consistency [Moh90], the rule condition is evaluated, the tuple is extracted from the page (assuming it qualifies), the page is unlatched, and the qualified tuple is returned.[1] This approach of acquiring a lock before evaluating the predicates has some serious drawbacks. Assuming that many transactions are concurrently inserting tuples into *stock*, the rule which is trying to read these inserted tuples has to wait frequently for write locks held by these inserting transactions. Furthermore, after the locks are granted, a large number of the tuples would be found to belong to non-*IBM* stocks and waiting for such locks would turn out to have been totally unnecessary.

To avoid this problem, we slightly extend the concurrency control protocols for such read accesses for active tables. The rule reads a tuple by latching the page, accessing the tuple, and evaluating the predicate without first obtaining a lock on the tuple. If the predicate is false, then the rule continues to the next tuple. If no qualifying tuple is found on the page, then the rule unlatches the page and continues to the next page. If the rule finds a qualified tuple, then it requests a lock on the tuple. This technique is an enhancement of the technique of postponing locking to the time after predicate evaluation, proposed in [Moh90]. With this enhanced method, the rule gains significant performance by avoiding unnecessary wait for locks on unqualified

[1] For further optimization, the page can be kept under the latch until a qualified tuple is found or the scan of this page is finished, whichever happens first.

tuples. Avoiding locking under these conditions is not only a concurrency advantage, but also a performance advantage because acquiring and releasing a lock costs hundreds of instructions.

The intuitive reasoning for why the above simple and effective scheme works is as follows. If a tuple qualifies, then we get a lock. This is what standard locking would have done. If a tuple does not qualify and it is not in the committed state, then two things can happen to it: either the changes will be committed, or the inserting transaction will rollback, thereby resulting in the tuple being eliminated. If the transaction were to commit subsequently, then our decision made without locking will turn out to be the same as the one that would have resulted from waiting for the lock: still the tuple does not qualify. If the transaction were to rollback, then there would not be any inconsistencies between the two decisions. Note that we need to consider only insert and undo of insert since update operations are not allowed on active tables' tuples.[2] Even in the case of tuples which qualify, we may be able to avoid locking by making use of the *Commit_LSN* idea presented in [Moh90].

As explained in Section 2, an active query may receive an EOF and later do *fetch_wait* and receive more tuples. Passive queries close their cursors once an EOF is encountered. Hence, we need to enhance DBMS operators for active queries. In Starburst, this problem is solved by decoupling the closing of cursors and return of EOF. Hence, low level operators may return EOF, and then when they are called again, they may return a tuple. For instance, the union operator returns EOF when all of its operands return EOF. Later on, when it is activated again, it does fetch on all of its operands, and returns any newly fetched tuple.

For *fetch-wait*, the *Alert* monitor must do efficient monitoring of data changes and filtering of the changes that are irrelevant to an active query. We do two levels of filtering: the first level filtering only deals with changes at the table level, whereas the second level filtering deals with changes at the tuple level. we now provide some filtering details.

An active query references a set of active tables, and only changes to these tables can affect this active query. The *Alert* monitor creates a Starburst attachment [LMP87] for each active table the rules refer to. The Starburst data manager sends a message to the *Alert* monitor whenever there is a change affecting such tables. The *Alert* monitor keeps, for each active query, a list of active tables that it refers to. For first level filtering, the *Alert* monitor uses this list to decide which active queries might be affected by a change to the database.

[2]We use this locking scheme only if the cursor is not part of a universal quantification, such as ALL subqueries. For universal quantification, lack of existence of a tuple is significant, hence we must know if the inserting transaction commits or rolls back before we can proceed. This is not a drawback since rules typically do not contain universal quantification.

The tuple-level filtering is more complex. Variations of RETE networks [For79] have been used in active DBMSs to perform this kind of filtering. Instead of extending RETE to apply in a concurrent/shared database environment, we are extending database indices to perform the task of prefiltering for active queries.

Let us consider an example. Suppose (1) we define an active query with condition *stocktype='IBM'* on *stock* and (2) we want to avoid doing a complete scan of *stock* to answer this query, then we may create an index on the column *stocktype* of *stock*. Whenever the DBMS inserts a tuple into *stock*, an entry will be added to this index. If this entry has *stocktype='IBM'*, then we can conclude that our rule might be affected. As stated earlier, active queries must see future appends. Let us see how an index manager can be extended to support this requirement. Suppose we use a variant of the B^+-tree index. When we insert a tuple into active table *stock* with *stocktype='IBM'*, we need to add the TID of this tuple to the *end* of the list of TIDs already present in the index that are associated with this key value. As a result, the active cursor on this index will see the new appends in the future. Further, when the cursor reaches the end of this list, it will stay there, so that it can read new TIDs when they are appended to the list. We can also use a hash index to achieve the same effect. Associated with a hash key value, there is a list of TIDs, and new TIDs must be appended to this list, similar to the B^+-tree case. Note that the index is *shared* among rules, active and passive queries.

We are investigating the extension of indices to support more complex predicates on active tables, such as range and join predicates. Postgres is also investigating a variant of B^+ trees to handle range predicates [KS91]. [HCKW90] discusses another approach for handling of range predicates; however, unlike our index, this index cannot be used for passive queries.

5 Related Work

This section briefly surveys the related work in this area and contrasts our approach with previous efforts. The HiPAC project [CBB+89] at CCA addressed several important issues and put forward several important results. The HiPAC work was not done in the context of a particular DBMS or a particular data manipulation language, and the results were not implemented in an integrated comprehensive system prototype. The HiPAC project used the relational model for the overall framework and the nested transaction model as the framework for the execution of rules. The paradigm of event-condition-action is used to express rules, where events can be built-in or user-defined. However, there is no such concept as set of events that one can refer to. In contrast, *Alert* has formalized events as *first class* tuples of active tables, which can be queried using the relational query language. Reuse of the relational language for this purpose has greatly simplified our event speci-

fication, particularly event composition, such as union and joins of active tables. HiPac introduces a special language to express event composition. HiPac keeps the changes made by built-in operations to database tuples in Δ relations (ΔRs), which are similar to our active tables for built-in operations on passive tables. However, ΔRs contain only the *net-effect* of changes. Furthermore, Alert emphasizes user-defined active tables, such as *journal* in our examples, where method activities are recorded. In this context, it is not clear what net-effect of changes means. For instance, the net-effect of *insert* followed by *update* is *insert*. But what is the net-effect of *Hire* followed by *promote*?

POSTGRES Rules System (PRS) is an integrated DBMS/rule system, as opposed to the layered approach of Alert. The first version of that system's design is described in [SHP88]. A second version (PRS II) [SHP89, SRH90] is still under development. Alert rules are based on operations on data (events), whereas PRS rules cannot refer to operations. This limitation of PRS has been addressed in PRS II, where the syntax for rules is more like that of HiPAC; however, events associated with only update, insert, delete operations can be specified. PRS and PRS II, unlike Alert, support only synchronous immediate triggers, and there is no explicit notion of transaction couplings. In PRS as well as PRS II, the mechanism used for rule firing is a tuple-marking algorithm in which special locks are acquired on tuples whose changes or retrievals would be relevant to one or more rules. When such data gets changed, then the modifying transaction notices the existence of the new types of locks and triggers the rules. In Alert, we have explored the idea of using indices on active tables for this purpose.

Another effort in the Starburst project has taken a somewhat different approach to supporting rules [WF90, WCL91]. Similar to Alert , it is based on SQL. Unlike Alert, however, rules can only refer to events associated with the built-in operations: update, insert, delete. Triggers are deferred and asserted at transaction commit time. The database changes are kept in transition logs similar to HiPac's Δ Relations or Alert's active tables for built-in operations. Unlike Alert, transition logs are not persistent since they are only used within a transaction and there is no support for continuation of transactions after a system crash. An in-memory transition log storage manager is used to maintain the transition logs. The transition log storage manager participates in transaction rollbacks. Similar to HiPac, there is a notion of net-effects. However, unlike HiPac, net-effects are not associated with transition logs, rather, they are associated with each rule. The net-effects of each rule are computed for each rule separately based on the last time the rule was considered for firing. Unlike Alert's layered architecture, an integrated rule system/DBMS solution is adopted. There is a built-in conflict resolver which works based on a user-specified partial order among the rules.

Commercial DBMSs have been introducing support for triggers, at various levels, for sometime, mainly due to customer needs for better support for integrity constraints. As a result, there has been a major effort in the SQL standard committee [ISO90] to support triggers and constraints. In the SQL standard, checking of constraints, such as salary> 0, or existence of a referential integrity constraint between departments and employees, is triggered by the DBMS. Users can specify whether constraints are to be checked at the end of each SQL statement or to be deferred and checked by an explicit command, similar to the *begin(end) assert* command of the Alert rule system. Support for triggers in the SQL standard is limited. The trigger events can only be built-in SQL operations (update, insert, delete) on a single base table. Triggers over views are not allowed. Triggers can only be part of the triggering transactions, and triggers cannot be nested.

Sybase supports triggers. Unlike Alert, only one trigger can be associated with an operation on a table. The action part of a trigger is limited to a sequence of SQL statements. Further, triggering is limited to one level, where the triggered actions themselves do not cause triggers to be fired.

KEE [IBM88], a commercial expert system, can interact with DBMSs using KEE Connection. However, unlike Alert, the emphasis is not on providing an integrated shared DBMS. KEE extracts data from the DB and builds a cache, and all the rules are applied to this cache. KEE provides a unified rule language for forward and backward chaining. Use of the unified active/passive query language in Alert is analogous to this. Forward chaining is analogous to Alert rules executed by triggering. Backward chaining is similar to using an Alert (nested) rule as a regular query, invoked by users. KEE rules are on instances of objects (tuples). Alert rules are set-oriented, in the sense that SQL is set-oriented, allowing (active) queries to deal with a set of tuples. KEE's the rule language does not have the closure property, support for rule nesting, and creation of rules on views. Triggering is based only on changes made by built-in operations. There is no notion of triggering based on invocation of methods.

6 Summary

We presented the design of Alert and its implementation in the Starburst relational DBMS. Alert is an extension architecture designed for transforming a passive SQL DBMS into an active DBMS. The salient feature of the design of Alert is the reuse to the extent possible the passive DBMS technology and minimal changes to the language and implementation of the passive DBMS. Alert provides a layered architecture that allows the semantics of a variety of production rule languages to be supported on top. Rules may be specified on user-defined as well as built-in database operations. Both synchronous and asynchronous event monitoring are possible.

The Alert approach of reusing passive DBMS technol-

ogy paid-off handsomely in its implementation. By having a rule language that is basically identical to the passive SQL, we reused almost all of the existing semantic checking, optimization, and execution implementations. By using active queries to specify rules and unifying active queries with passive queries, we developed a rule language that inherits the rich set of SQL constructs to specify arbitrarily complex rule conditions involving multiple tables, nesting of query expressions, and particularly the closure property. By modeling events as tuples in active tables, we reused most of the storage management of regular tables to keep track of sets of events. We could use database indices and query optimization techniques for event detection. We could also translate large body of concurrency control and recovery knowhow for contention reduction and use of production rules in a shared environment.

Several issues need to be further explored in the context of our approach to adding active DBMS capabilities to a passive DBMS. These include extension of efficient monitoring of changes using indices, particularly for range predicates, concurrency control issues involving already executing transactions and rule activations, multi query optimization of active queries associated with rules, and parallel execution of conditions and actions.

7 Acknowledgements

We would like to acknowledge George Lapis for implementing the DDL for active tables and creating attachments for Alert. John McPherson's insight in defining Starburst operator protocols so that they can continue after returning EOF made Alert feasible. Bruce Lindsay and John McPherson implemented the union operator that made it possible to specify Alert rules involving union. We would also like to thank Guy Lohman for many useful discussions on the Alert concepts and for his suggestions for some of the examples of Alert rules presented in this paper. Thanks are also due to Laura Haas and Guy Lohman for their comments on an earlier version of this paper.

References

[ADL91] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. *Polyglot: An Object-Oriented Type System for Multi-Language Support.* Technical Report, IBM Almaden Research Center, 1991. under preparation.

[BW77] D. G. Bobrow and R. Winograd. *An Overview of KRL, a Knowledge Representation Language.* Cognitive Science, 1:3-46, 1977.

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

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ladin, D. McCarthy, A. Rosenthal, S. Sarin, M. Carey, M. Livny, and R. Jauhari. *The HiPAC Project: Combining Active Databases and Timing Constraints.* ACM-SIGMOD Record, 17(1):51-70, March 1988.

[For79] C. L. Forgy. **On the Efficient Implementation of Production Systems.** PhD Thesis, Department of Computer Science, CMU, February 1979.

[For81] C. L. Forgy. *OPS5 User's Manual.* Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[GLPT76] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. *Granularity of Locks and Degrees of Consistency in a Shared Database.* In G. Nijssen (Ed.), **Modeling in Data Base Management Systems**, pages 365-394. North-Holland, Amsterdam, 1976.

[HCKW90] E. Hanson, M. Chaabouni, C. Kim, and Y. Wang. *A Predicate Matching Algorithm for Database Rule Systems.* In **Proc. ACM-SIGMOD International Conference on Management of Data** [Pro90a].

[HCL+90] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. *Starburst Mid-Flight: As the Dust Clears.* **IEEE Transactions on Knowledge and Data Engineering**, pages 143-160, March 1990.

[HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. *Extensible Query Processing in Starburst.* In **Proc. ACM-SIGMOD International Conference on Management of Data** [Pro89], pages 377-388.

[HLM88] M. Hsu, R. Ladin, and D. McCarthy. *An Execution Model for Active Data Base Management Systems.* In **Proc. 3rd International Conference on Data and Knowledge Bases - Improving Usability and Responsiveness** [ICD88].

[IBM88] IBM. **IBM Knowledge Engineering Environment/370 (KEE/370), User's Guide, Release 1**, Document No. SC26-4540, December 1988.

[ICD88] Proc. 3rd International Conference on Data and Knowledge Bases - Improving Usability and Responsiveness, Jerusalem, June 1988.

[ISO90] ISO_ANSI. *ISO-ANSI Working Draft: Database Language SQL2 and SQL3; X3H2/90/398; ISO/IEC JTC1/SC21/WG3*, 1990.

[KS91] C. Kolovson and M. Stonebraker. *Segmented search trees and their application to databases.* Technical Report, 1991. under preparation.

[LMP87] B. Lindsay, J. McPherson, and H. Pirahesh. *Data Management Extension Architecture.* In Proc. ACM-SIGMOD International Conference on Management of Data, pages 220–226, San Francisco, May 1987.

[Mar90] V. Markowitz. *Referential Integrity Revisited: An Object-Oriented Perspective.* In Proc. 16th International Conference on Very Large Data Bases [Pro90b].

[MD89] D. McCarthy and U. Dayal. *The Architecture of an Active Database Management System.* In Proc. ACM-SIGMOD International Conference on Management of Data [Pro89].

[Moh90] C. Mohan. *Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems.* In Proc. 16th International Conference on Very Large Data Bases [Pro90b]. Also available as IBM Research Report RJ7344, IBM Almaden Research Center, February 1990.

[PMC+90] H. Pirahesh, C. Mohan, J. Cheng, T. Liu, and P. Selinger. *Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches.* In Proc. 2nd International Symposium on Databases in Parallel and Distributed Systems, Dublin, July 1990.

[Pro89] Proc. ACM-SIGMOD International Conference on Management of Data, Portland, May-June 1989.

[Pro90a] Proc. ACM-SIGMOD International Conference on Management of Data, Atlantic City, May 1990.

[Pro90b] Proc. 16th International Conference on Very Large Data Bases, Brisbane, August 1990.

[RCBB89] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakely. *Situation Monitoring for Active Databases.* In Proc. 15th International Conference on Very Large Data Bases, Amsterdam, August 1989.

[SHP88] M. Stonebraker, E. Hanson, and S. Potamianos. *The POSTGRES Rule Manager.* IEEE Transactions on Software Engineering, 14(7):897–907, July 1988.

[SHP89] M. Stonebraker, M. Hearst, and S. Potamianos. *A Commentary on the POSTGRES Rules System.* ACM SIGMOD Record, 18(3):5–11, September 1989.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. *On Rules, Procedures, Caching and Views in Data Base Systems.* In Proc. ACM-SIGMOD International Conference on Management of Data [Pro90a].

[SLR88] T. Sellis, C.-C. Lin, and L. Raschid. *Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms.* In Proc. ACM-SIGMOD International Conference on Management of Data, pages 404–412, Chicago, June 1988.

[SRH90] M. Stonebraker, L. Rowe, and M. Hirohama. *The Implementation of POSTGRES.* IEEE Transactions on Knowledge and Data Engineering, 2(1), March 1990.

[Syb87] Sybase, Inc. Transact-SQL User's Guide, 1987.

[TPC89] TPC benchmark group. *TPC Benchmark, A Draft 6-PR Proposed Standard*, 1989. Available from ITOM International Co., POB 1450, Los Altos, CA 94023.

[WCL91] J. Widom, R. J. Cochrane, and B. G. Lindsay. *Implementing Set-Oriented Production Rules as an Extension to Starburst.* Research Report RJ 7979, IBM Almaden Research Center, February 1991.

[WF90] J. Widom and S. Finkelstein. *Set-Oriented Production Rules in Relational Database Systems.* In Proc. ACM-SIGMOD International Conference on Management of Data [Pro90a], pages 259–270.

[ZB90] D. R. Zertuche and A. P. Buchmann. *Execution Models for Active Database Systems: A Comparison.* Technical Memorandum TM-0238-01-90-165, GTE Laboratories, 1990.