CHAPTER 4

4. DESIGN AND DEVELOPMENT OF
RULE SCHEDULER

4.1 RULE SCHEDULING MECHANISM

In active ORDBMS, an application defines rules and specifies the desired behavior of each rule in the form of ECA. In an application, the user specifies the rule definition associated with event, condition and action, priority and the coupling mode. Once a set of rules is defined, the active database system monitors the occurrences of events pre-specified by ECA rules. Whenever a specified event occurrence causes the multiple triggering of rules at the same time, the rule conflict occurs and thus rule scheduler determines the order and the execution semantics: sequential or concurrent.

The sequential rule execution enforces a single rule execution at a time and supports that rules have to be in a predefined sequence of execution order enforced by priorities. Also sequential approach does not use concurrency or maximize performance. However, concurrent rule execution improves system performance and thereby reducing the overall execution time for rule transaction. Concurrent rule execution speeds up the ADBS rule execution process for improving the system performance. Thus in the presence of multiple rules, the rule scheduler need to support serial execution of rules, concurrent execution of all rules or a combination of both based on the static priority scheme in active ORDBMS. When a rule is triggered, the time to start the execution of that rule is determined by the semantics of the coupling mode involved. Thus, when multiple rules are triggered together, the order of execution among those rules is determined by assigning priorities and rules are executed based on their coupling modes.

The two important factors considered for designing the rule scheduler in this research is the priority and coupling mode. In this research work, the scheduler uses numeric user-defined priority scheme for ordering the rules. Rules are placed in order using a numerical scheme in which each rule is given an absolute value and the highest priority value of a rule is 1. The user defines priorities at the time of rule
definition. Rule systems are not static and thus rules are continuously added and deleted, and user-defined priorities between existing rules are altered. Thus user-defined priorities are dynamic – they may be dropped and added at any time during the existence of rule set. Priorities between rules are used to refine termination analysis. User-defined priorities override default priorities and it allows the user to specify absolute priority values to rules. User-defined priority system is used to determine the order of rule execution. Thus, user-defined priorities provide better opportunity to define and execute rules based on the requirement of the application than system defined priority.

The coupling mode specifies the time when a triggered transaction is executed with respect to the triggering transaction or an event. If the rule is in immediate coupling mode, then the triggered transaction executes immediately and the triggering transaction has to wait until the completion of the triggered transaction. However if the rule is in deferred mode, then triggering (main) transaction continues and the triggered rule has to wait for the completion of the triggering transaction. This work supports rule execution with immediate and deferred coupling modes. If the priority and the coupling mode are not specified by the user, the default priority and coupling mode are taken into consideration. The default priority is the lowest priority and the default coupling mode is immediate.

The rule scheduler uses the user-defined priority information to execute the multiple triggered rules concurrently with the same priority and sequentially with different priorities based on the coupling mode. The scheduler allows rules with the highest priority to execute first and after their completion, it allows the next rule with lower priority to execute and the algorithm for rule scheduling is shown in Figure 4.1. Based on the algorithm the scheduler has to keep track of all the rules that have been triggered for execution.

Whenever an event triggers many rules with different priorities and coupling modes at the same time, the various steps taken into account are: The scheduler has to segregate the rules based on their coupling modes because immediate rule has to be executed first even if a deferred rule has a higher priority. After the segregation, the scheduler has to suspend the main (triggering) transaction. The scheduler uses the thread functions to suspend and continue rule processing. The triggering rule and the
triggered rules (subtransactions) are all executed using multi-threaded operations. Thus the scheduler has to keep track of all rules that have been triggered and spawn the rules one after the other based on their priority.

Step 1: Check whether all the rules \( R_1, R_2, R_3 \ldots R_n \) belongs to \( D \), where \( D \) is a domain.

Step 2: If there is only one rule \( R_i \) is raised by an event \( E \), then print “No rule conflicts”; go to step 8. Otherwise, go to next step.

Step 3: If an event \( E \) triggers multiple rules \( R_1, R_2, R_3 \ldots R_n \) then print “Rules are conflict” \( R_p = \{\text{empty}\} \) initially, form a process-rule-list \( R_p = \{R_1, R_2 \ldots R_n\} \).

Step 4: Check the priority number of each rule in \( R_p \). If the priority of each rule is different, then print “Sequential scheduling”; go to step 7. Otherwise go to next step.

Step 5: If the priority of the rules in \( R_p \) are equal then print “Concurrent scheduling”; and execute all rules concurrently; go to step 10. Otherwise go to next step.

Step 6: If the priorities of the rules in \( R_p \) are equal and different then print “Hybrid scheduling”; and execute the rules both concurrent and sequential; go to step 10. Otherwise go to step 9.

Step 7: Execute the rules in \( R_p \) sequentially one by one based on the highest priority until the \( R_p \) set is empty; go to step 10.

Step 8: Check the condition of \( R_i \); If \( R_i \) = true then print “Ready for execution”; and execute the rule. Otherwise go to step 10.

Step 9: If the rules are not prioritized, then execute the rules in \( R_p \) sequentially one by one based on the proposed system priority.

Step 10: Exit rule processing and resume the transaction.

**Figure 4.1 Rule scheduling algorithm**

The scheduler also has to keep track of the stage of execution of each rule based on the four operating modes which includes ready – when the rule is triggered and is ready to execute; wait – when the rule triggers another rule and is waiting for that rule to finish execution; executing – when the rule is currently executing; finished – when the rule has finished processing. Based on this knowledge, the scheduler decides when to allow each rule to execute.
**Architectural design of the rule scheduler**

Active database systems have been implemented as extensions to an underlying DBMS based on the client/server architecture principle. Object-relational database systems are implemented as a fat server which means that event detection and trigger execution is carried out on the server side. When building an active database system a decision has to be made about the type of architecture used for integrating active functionality into the database systems.

Active rules are triggered in the context of transaction execution and to maintain the correctness of the operation, the ADBS rule execution model uses the transaction management mechanism of the underlying DBMS for rule scheduling and execution using an integrated architecture. This research work uses the integrated architectural design to entail active behavior into underlying DBMS [Ramakrishnan and Gehrke, 2000] with rule processing modules as shown in Figure 4.2. The major component extensions incorporated for providing active capability into ORDBMS include the event detector, rule manager and rule scheduler.

Transaction manager ensures that transactions request and release locks according to a suitable locking protocol for the execution of user transactions. The role of the transaction manager is to manage and control user transactions and it communicates with the lock manager, responsible for implementing a particular strategy for concurrency control. Transaction manager is also responsible for supporting nested transactions used for concurrent execution of rules.

Nested transactions play an important role in achieving concurrency within a transaction. A nested transaction is a hierarchy of subtransactions, where each subtransaction may contain other subtransactions. The nested transaction model is a suitable tool to implement rule execution in ADBS since it provides a good model for concurrent rule execution, whenever an event triggers multiple rules. Thus, in this research work nested transaction model is used for modeling concurrent rule execution and it handles multiple triggering of rules that occurs when an event causes more than one rule to be triggered. Concurrent execution of rules can also bring along conflict with respect to data access and the nested transaction synchronization scheme is used to handle the conflicts with locking protocol.
The lock manager which keeps the track of requests for locks and grants locks on database objects when they become available. The lock manager handles concurrency control issues while executing concurrent rules using nested transaction model. The data structure used by the lock manager is hash table that serves two
different purposes, i) given an object which are uniquely identified, to find out all the
transactions that hold a lock on that object, ii) for a transaction that is uniquely
identified, to find out all the objects that holds a lock. The hashing is done on the
object with the object-id and storage group number. Each object has an anchor node
and if more than one object hashes to the same bucket then a linked list at that bucket
is maintained. Also for each access of that object by a transaction, a separate node is
created which is anchored on the object node. Thus a separate linked list from each
anchor (object) is formed for all transactions that hold a lock on that object. A list of
all transactions is maintained with links nodes in the anchored hash table with the
same transaction-id are connected as a list and this is helpful in tracking the
transaction with all objects held by it.

Event detectors responsible for detecting and reporting primitive events such
as database operations or application defined events to the rule manager. When an
event is detected by an event detector it is reported to the rule manager. The rule
manager receives the notification of message raise event from an event detector with
the details such as the type of the event, rule(s) associated with that event. The rule
manager then retrieves the appropriate event from the database and determines the
rules that are associated to the primitive event. The rules associated with the detected
events are evaluated and stores the rules whose conditions are evaluated to true in a
conflict set and then calls the transaction manager to create a transaction for the
conflict triggered rules for scheduling and execution.

The rule scheduler is responsible for handling rule conflicts when multiple
rules are triggered by an event at the same time. The rule scheduler is activated by
getting a notification message about the conflict triggered rules raised by an event
from the transaction manager with the support of nested transaction model for
executing rules concurrently. In the presence of multiple rules, the architecture need
to support prioritized serial execution of all rules, concurrent execution of all rules or
a combination of both. The rule scheduler implements the multi-threaded mechanism
for the prioritized concurrent and sequential rule execution.
4.2 MULTI-THREADING SYSTEM

From the above proposed rule scheduling mechanism, the scheduler has to keep track of all rules that have been triggered after detecting the occurrence of an event and multiple triggered rules are executed based on their priority and coupling mode. In order to achieve this function, multi-threading mechanism is used in this research work. Multi-threading is a widespread programming and execution model that allows multiple threads to exist within the context of a single process. These threads share the process resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines, because the threads of the program naturally lend themselves to truly concurrent execution.

Thread is a sequence of instructions executed within the context of a process. Within a single thread, there is at any instance a single point of execution and in the case of multiple threads that any instant the program has multiple points of execution. That is, the concept of thread that allows multiple threads of execution in a single process are called multi-threading where as a single thread of execution per process is called as single threading. A thread is an independent execution path, able to run simultaneously with other threads. Thread facilities are often referred as light weight processes since thread creation, existence, destruction and synchronization are cheap and the developer can use them based on their needs. With the help of threads the response time of the application has been increased. The advantages of using threads are: i) using threads multiple activities can be performed simultaneously ii) achieve faster computations by doing two different computations in two threads instead of serially one after the other iii) use of threads saves wastage of CPU cycle and increase efficiency of an application.

Multi-threading separates a process into many execution threads and each of which runs independently. By using a light-weight multithreading facility, concurrent operations are performed [Birrell, 2005]. Multi-threading has several potential benefits includes enhanced performance, increased throughput and greater user
responsiveness. Performance is defined as the total elapsed time of a given computation or set of computations. Throughput or response refers to the average turnaround time for each computation. Multithreading takes less time to terminate a thread and it takes less time to do control switching between two threads within a same process. Multi-threading makes it easier to communicate between different execution traces.

Object-oriented programming environment such as Java, C# provides support for concurrent programming through multithreading. A multi-threading facility allows for concurrent operation with multiple simultaneous points of execution. In order to achieve the mechanism of a rule scheduler for concurrent and sequential rule execution, this research work uses the concept of C# multi-threading system. C# is modern, general purpose and multi-paradigm programming language enclosing object-oriented, imperative, functional, generic, event-driven and component oriented programming styles developed by Microsoft and first released in July 2000 [Rabah et al., 2010].

4.3 DEVELOPMENT OF RULE SCHEDULER

In order to achieve the static priority based concurrent and sequential rule execution, the rule scheduler implements the concept of multi-threaded mechanism. Multi-threading system is used for performing concurrent rule execution to increase the function of the rule scheduler of an active ORDBMS. The important feature required by the scheduler is the ability to start and stop threads (rules) and this feature is achieved by using thread functions. Each rule triggered during the execution of an application has to be a separate thread. The execution of the thread is controlled by the scheduler depending on its priority and coupling mode.

Threads support concurrent operations and thread execution is controlled after the creation of the thread with various methods. The various thread functions used in this research work are:

- thrd_create – creates a new thread.
- thrd_self – returns the thread-id of the calling thread.
• thrd_suspend – blocks the execution of the thread specified by its id.
• thrd_continue – unblocks the blocked thread.
• thrd_join – wait for the termination of a thread(s).
• thrd_kill – sends signal to a thread.
• thrd_yield – causes the calling thread to yield execution to another thread.
• thrd_abort – terminates the execution of the thread.
• mutex_lock – locks a mutex variable to support synchronization among threads whenever multiple threads accessing the shared data.
• mutex_unlock – unlock the locked mutex variable.

The various parameters that are to be known when forming a thread to execute a rule are parent rule-id, priority and coupling mode. The data structure named as process-rule-list that is used to store the information includes the parent rule-id, priority and coupling mode that are obtained when creating a thread for that rule. The rule-id of the rule is the thread’s id that is obtained when creating a thread for that rule. The system assigns a unique-id to every thread it creates. Rules triggered by an event are the subtransactions of the main (triggering) transaction which have the parent-id as 1. The priority and coupling mode are obtained from the rule as they are provided during rule creation. Also an operating mode is associated with each rule to capture the state of each rule execution. The four states of operating mode are: Ready, Wait, Executing and Finished. The finished operating mode is used by the scheduler to make sure that all the rules in the immediate coupling mode have finished executing.

In an application, the user specifies the rules with priority, coupling mode and their respective procedures. The application proceeds normally until it reaches a procedure call that is detected by an event. The overview of thread implementation is shown in Figure 4.3. When an event is signaled, the event detector detects which of the primitive event is occurred. Once an event is detected, a set of rules triggered are as shown in the rule processing stage of Figure 4.3. Every event detected in an
application is wrapped with a Notify call that creates a thread for each rule. For each qualifying rule, a separate thread is created and inserted into the process-rule-list. Then the Notify calls the function Process_imm-rules. The function Process_imm-rules activates the scheduler and the rules are executed based on the priorities and coupling modes. The function then looks at the process-rule-list and finds the triggered rules in the immediate coupling mode based on the current thread-id.

![Figure 4.3 Overview of thread implementation](image)

When the scheduler activates a rule, it wakes up the thread associated with the thread-id in the process-rule-list. Then the thread checks the condition of the rule and executes the action associated with the rule based on the priority if the condition evaluates to true. The scheduler is activated and the function Process_imm-rules waits until all the triggered rules in the immediate coupling mode complete its execution.
In order to process deferred rules, the primitive event Notify_def-rules is used and this triggers a pre-defined rule immediately and the action part of this rule calls the function Process_def-rules which indicates the scheduler to process deferred rules as shown in Figure 4.3. The Notify_def-rules event waits for the completion of all deferred rules and a flag variable is used to block the parent transaction from commit until all the rules spawned complete their execution. When a rule thread completes its execution the corresponding rule is removed from the process-rule-list and at the end of the top-level transaction, the process-rule-list is empty. The transaction commits and the application continues further when the process-rule-list has no more rules to execute.

The scheduler runs in loop acting on the process-rule-list and the scheduler is suspended and awakened by the thread in the system. The scheduler starts at the head of the process-rule-list. If the head points to a rule with priority -1 means that all rules that are triggered within that transaction have completed execution and the scheduler then sends signal to the triggering transaction informing it of the status.

The scheduler operates on rules in the process-rule-list based on its operating mode. The operating mode Ready indicates that the rule is ready to execute, the scheduler awakens the thread allowing rule execution. The highest priority rules which have the operating mode as Ready are fired and the operating mode is changed to Executing. The operating mode Wait indicates that the thread is on wait, which means that certain rules need processing and the operating mode Finish represents the completion of the rule processing of that rule.

The psuedocode for rule scheduling and execution using threads and how the thread function is grouped is shown in Figure 4.4. As detailed above, when a primitive event is signaled, the event detector notifies the occurrence of the primitive event and determines the triggered rules for execution. Whenever a rule is triggered in immediate coupling mode, a thread is created with the thread-id from a pool of threads and transforms the function which checks the condition and performs the action to a thread with the appropriate priority. Once all the immediate rules are in the form of threads, the main application is suspended and the rule scheduler is activated. After the execution of all triggered rules, the triggering transaction resumes execution.
Since a deferred rule is executed only when Notify_def-rules primitive event is signaled, hence deferred rules are treated in the same way as immediate rules.

```c
Initiate_thread()
{
  If(there is rule to be scheduled)
  {
    If (thread is available)
    {
      priority = assign_priority;
      thread_id = get_thread();
      create_thread(thread_id, priority, cond_action)
    }
  }

  //condition and action grouped as the body of the thread
  cond_action(rule_cond, rule_action)
  {
    void (cond)( ) = rule_cond;
    void (action)( ) = rule_action;
    sub= begin_subtransaction(current)
    if ((cond)( ) == true )
      (action)( );
    end_subtransaction(sub);
  }
```

Figure 4.4 Psuedocode for rule scheduling and execution using threads

An example scenario of the process-rule-list after inserting rules for the execution of multiple triggered rules is shown in Table 4.1. When multiple rules are triggered by an event in the application, rules are inserted into the process-rule-list based on the coupling mode, priority and its parent. The process-rule-list is divided into two parts, one for rules with immediate coupling mode and other for rules with deferred coupling mode. If the rule has been triggered from within the action of another rule then the parent rule is searched and the new rule is placed after the parent based on the coupling mode in the process-rule-list. The operating mode of all new rules inserted into the process-rule-list is in Ready state. In case of rules with
immediate coupling mode when a rule is inserted, the parents operating mode is changed to Wait state. The parent rule has to wait for the termination of all child rules before it can proceed.

Table 4.1 Process-rule-list

<table>
<thead>
<tr>
<th>Triggered Rules</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coupling Mode</td>
<td>I</td>
<td>I</td>
<td>D</td>
</tr>
<tr>
<td>Priority</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Parent</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In Table 4.1, the rules that are triggered by an event in the application are R1, R2 and R3. The parents of the triggered rules are given the value 1. This is the default id of the main ( ) thread and indicates that the rule has been triggered from the main thread. R1 and R2 are rules that are in the immediate coupling mode and R3 has a deferred coupling mode. The rules R1, R2 and R3 are placed in the list based on their priorities and coupling mode. Rules R1 and R2 both have same priority and are in immediate coupling mode.

Rule R3 is placed at the end of immediate rules since it is in deferred mode. Thus the main transaction waits for the completion of R1 and R2’s execution from which they are triggered. The main transaction then continues and at commit time, it allows R3 to execute. The rules in the process-rule-list are executed based on the rule scheduling algorithm. The developed rule scheduler executes the rules R1 and R2 concurrently because of its equal (same) priority value and rule R3 has been executed after the completion of R1 and R2.
4.4. SUMMARY

In this chapter, rule scheduling mechanism with algorithm for handling triggered rule conflicts in active object-relational database systems has been proposed. The integrated architectural design to entail active behavior into ORDBMS for rule processing with the modules such as rule manager and rule scheduler has been described. The rule manager receives the notification of message raise event from an event detector and retrieves the appropriate rules that are associated to the detected primitive event for condition evaluation. The rule manager stores the rules whose conditions are evaluated to true in a conflict set. The rule scheduler is responsible for handling rule conflicts when multiple rules are triggered at the same time by an event. The developed rule scheduler implements the C# multi-threading system for concurrent rule execution and the pseudocode for rule scheduling and execution using threads has been specified.