Experiments on the Concurrent Rule Execution in Database Systems

Alex Delis, Louisa Raschid and Timos Sellis
University of Maryland Institute for Advanced Computer Science (UMIACS)
University of Maryland
College Park, MD 20742

Abstract

In this paper we study issues pertinent to the concurrent execution of rules in a DBMS system. We model rules as database transactions. As such, they should follow serializability as their correctness criterion for execution. Rule execution has the additional constraint that their conditions must be true in the database for the actions to execute, and rules must fail when their conditions are not true any longer. Based on this observation, two locking based protocols are discussed. Information on the possible conflicts between conditions and actions of rules is used to provide greater concurrent access to the relations, based on a new lock paradigm. A simulation testbed was developed in order to study the rule features and database characteristics that play an important role in the performance of concurrent production rule execution.

1 Introduction

It is expected that rule-based reasoning is going to become an integral part of the next generation database systems [3, 10]. Rules are used to specify operations that need to be applied as soon as certain conditions qualify. The conditions are created by changes on the database. Production rules are a versatile mechanism used to maintain derived data, integrity constraints, triggers, alerters [10], as well as other applications such as knowledge-intense systems, engineering information systems, and expert systems [4]. Although semantics and algorithms for the processing of rules in such systems have been proposed [7], in all cases, processing of rules is done in a strictly sequential fashion. Given the fact that rules systems are to be run on top of commercially available database systems, it is very important that concurrent execution of rules is exploited in order to maintain high throughput. The goal of this paper is to present alternative techniques for concurrent rule execution and to evaluate them using a simulation testbed.

Rules have the general format: IF <CONDITION> THEN <ACTION>, where <CONDITION> is a set of queries over the database and the <ACTION> part of a rule specifies a sequence of primitive operations, such as data insertions and deletions, to be performed on the database when the <CONDITION> is satisfied. Efficiency is of major concern when dealing with large rule bases. In a DBMS environment, a very important issue is the greater accessibility of the disk resident database during execution.

So far, in rule-based AI applications, the data is main memory resident and the production rule execution is similar to the execution of a single program with exclusive access to main memory data. In contrast, a DBMS always supports multiple user access to a large, disk resident database. The operations of each user are encapsulated within a transaction, which has the property of indivisibility. Since transactions may update the database, protocols are needed to maintain a correct execution, where the transactions do not interfere with each other. It is imperative to define a correctness criterion for the concurrent execution of rules in the presence of conflicting page requests. In general, a rule consists of a query corresponding to the condition which must be satisfied by the database, followed by the actions to be executed. The execution of the actions can change the database so that the condition of some other rule(s) is no longer true. Consequently, when a rule determines that its condition is no longer true, it must fail. Thus, the serializability criterion [1] for rules is re-defined on the basis of conflicts between the conditions and actions of the rules [6].

The organization of this paper is as follows: in section 2, we outline the execution of rules in both traditional AI main-memory based systems and in DBMSs and review two lock management techniques. In section 3, we present a simulation testbed and describe some of our experiments. Conclusions can be found in the last section.
2 Execution of Rules

In this section, we describe the execution of rules in both traditional AI main memory oriented systems and DBMSs. In AI systems, the rule-system repeatedly performs the following operations until no more rules can be executed: 

- **Select**: Select one rule out of the conflict set; if there is no such rule, halt. 
- **Act**: Perform the actions in the RHS of the selected rule. This will change the content of the database, and, possibly, the conflict set.

All qualifying rules are entered in the Conflict Set; this holds the set of rules that are currently applicable, and are ready with their actions to update the database. Each element of the Conflict Set is a rule instantiation and includes the rule as well as a single combination of tuples that satisfies the condition of the rule. The Rete Match algorithm [5] gave an early solution to the problem of efficient rule matching in the Conflict Set. Another important issue in CS is the process of selecting rules which may have a serious impact on performance. This is particularly pertinent to DBMS environments where rule related data are disk resident.

The architecture for a relational DBMS implementation of a rule sub-system is illustrated in Figure 1. Two main components execute simultaneously and their tasks are as follows: 

- Selecting rules whose conditions are satisfied in the database, i.e. these rules are applicable and may be executed. This is the Match/Maintenance Process (MMP). 
- The second task is that of the concurrent execution of selected rules. This is the Concurrent Execution Process (CEP) in Figure 1.

The Match component is the same as discussed before. We allow the Match and Maintenance process to capture some auxiliary and sometimes redundant information. Using this auxiliary information (AUX) [8, 9] makes the Match phase much faster, as it need not check conditions of all rules, but instead, only a small subset. The Maintenance process is able to monitor the database updates and the Match is done more efficiently, in an incremental fashion.

Changes in database relations trigger the Match/Maintenance Process (MMP), reflected by the arc labeled DataRead. The MMP identifies rules whose antecedents are satisfied and are ready for execution. These rules, together with identifiers for the tuples of the relations that satisfy the conditions, are placed in the Concurrent Execution Set (CES) (reflected by the arc CESUpdate). The MMP also executes appropriate actions to maintain the auxiliary data consistent with the content of the database, shown by the operation AuxUpdate.

It is with respect to the Select and Act phases of the execution cycle in AI systems, that our DBMS implementation differs significantly. An alternative to the AI conflict resolution strategies is considered here, which is of practical importance in a DBMS environment. This allows several qualifying rule instantiations to execute concurrently and lets the concurrency control manager to take care of concurrent accesses to the same data by the rules. We note that concurrent execution strategies discussed here may not have the property of repeatability that is associated with many AI conflict resolution strategies. Thus, there is a trade-off between repeatable execution on the one hand and concurrent execution with greater availability of the database and support of multiple users.

In this DBMS setting, each instantiation of a rule is considered to be executing within an individual transaction by the Concurrent Execution Process (CEP). Within the CEP, all the tasks associated with the execution of a candidate rule instantiation will be defined as a single transaction. Within such a transaction, one task is retrieval(s) from the database, to obtain the tuples that caused the rule instantiation to become applicable for execution. This is reflected by the arc labeled DataRead of the CEP. The other task is executing the corresponding actions. These actions represent changes to the database and include insertions, deletions (and updates) of tuples. This is reflected by the arc labeled DataUpdate.

Recall that executing the actions of a production rule makes changes to the database. Consequently, the contents of the auxiliary relations (AUX) must be updated; this changes the set of rules whose conditions are now satisfied. With a serial execution strategy, this is generally not a problem. However, with a concurrent execution strategy, there is a possibility that changes made to the database by a rule that completes execution and commits, affect the data that is required by another rule that is executing concurrently. It is important to identify how the actions of a rule that commits affect other rules, and define a correctness criterion for the concurrent execution. This is considered next.

Figure 1: Rule Execution in a DBMS Rule-Subsystem
2.1 The Locking Protocol

In this subsection, we give some observations for the concurrent rule execution that led us to the formulation of two locking protocols: one based on the standard two-phase locking (2-φ) (CONV) and another extended one (NEW). These observations specify when locks must be obtained for checking conditions and executing actions, and also determine when transactions must be aborted (when the corresponding rules fail). Locking granularity is at both the relation level and the tuple (page) level.

- Each transaction \( T_i \) must obtain an RL (Read Lock) for specific tuples (or specific pages) of the relations on which it positively depends; these tuples are used to verify the condition of the rule. Obtaining these locks prevents the subsequent deletion or update of these tuples, by other transactions.

- Each transaction \( T_i \) must obtain an RL for the relation(s) on which it has a negative dependency. It must subsequently obtain an RL on all the tuples (pages) of the relation and verify that there are no tuples satisfying the negative dependencies in the condition of the corresponding rule.

- Each transaction \( T_i \) must obtain an exclusive WL (Write Lock) for specific tuples (pages) of the relation that it deletes (or inserts into). In the case of the deletes, these would be tuples satisfying a positive dependency, for which it would have previously obtained an RL. Consequently, these tuples exist at the current time, and would not have been deleted by some previously committed transaction. In the case of the inserts, a WL is obtained for the page into which the tuple is inserted.

- Each transaction \( T_i \) must obtain a WL for the relation(s) into which it inserts tuples. In this case, a previously held RL on the relation may be converted into an exclusive WL, if no other transaction is holding a RL on relation \( R \).

- During its execution, a transaction may be delayed because it cannot obtain a particular lock. If it is indefinitely delayed due to a deadlock situation, then all the locks currently held must be released, all requested locks must be canceled and the transaction must be terminated and restarted. A transaction which successfully obtains all locks and completes execution can commit all its changes to the database.

The two phases are still preserved. Once all the required locks are obtained, a transaction can execute. If it executes successfully, it will commit all of its changes to the database before releasing all locks. If it fails, the transaction must be aborted (i.e., release of all locks and no changes in the database). A more rigorous discussion on these observations can be found in [6].

2.2 Lock Management

For a conventional lock manager based on Read–Write conflicts, there is a hierarchical dependency between locks at the page level and the corresponding relation level, i.e., when a relation is locked, then all of its pages are effectively locked. This is needed to prevent problems such as phantom inserts. The 2-φ protocol (CONV) uses the following conventional lock compatibility matrix to grant locks.

<table>
<thead>
<tr>
<th></th>
<th>RL-Page</th>
<th>WL-Page</th>
<th>RL-Rel</th>
<th>WL-Rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL-Page</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>WL-Page</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>RL-Rel</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>WL-Rel</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

A RL or a WL may be obtained, on individual pages or on the entire relation. RLS are shared locks but, there can only be a single WL held on a page or relation (exclusive lock). A transaction that holds an RL, either at the page level or the relation level, is allowed to obtain a corresponding WL, provided no other transaction holds an RL. To explain the hierarchical dependency between relation and page level locks, a RL on a page is compatible with a RL on the corresponding relation. However, it is incompatible with a WL on the corresponding relation or page of that relation. Similarly, a RL on a relation is compatible with another RL on that relation or with a RL on pages of that relation. It is incompatible with a WL on that relation or a page of that relation.

When we move away from the Read–Write conflicts and consider the conflicts between the conditions and actions, we can use the information on the rule instantiations that do not conflict to obtain a new lock compatibility matrix. This matrix allows pages of a relation to be read, and even updated, while there is a lock being held on the relation itself. The following is the NEW lock compatibility matrix:

<table>
<thead>
<tr>
<th></th>
<th>RL-Page</th>
<th>WL-Page</th>
<th>RL-Rel</th>
<th>WL-Rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL-Page</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>WL-Page</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>RL-Rel</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>WL-Rel</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

The upper left quadrant and the lower right quadrant of the matrix are identical to the conventional matrix. However, the upper right and lower left quarters of this matrix are different. With the NEW matrix, a RL on a relation is also compatible with a WL on any page of that relation. Similarly, a WL on the relation is compatible with both a RL or a WL on any page of that relation. In other words, locks at the relation level do not affect locks at the page level and the two levels or granularities of locks are independent of each other. This is a very attractive feature; since there is no hierarchical dependency between relation and page level locks, relation level locks can be represented by a lock on a header page, corresponding to that relation. The relation level locks are now identical to page level locks on the header page. As a result, the task of granting a lock to a transaction at either level, and also detecting deadlock among a group of transactions, is simplified considerably. In [6], we have proved that our original protocol in conjunction with this new lock compatibility matrix, produce a serializable concurrent execution.
3 Simulation Model and Results

The performance of the concurrent execution of rules is studied using a simulation DBMS testbed. Our initial effort was to identify characteristics of rules, features of the database environment, and elements of lock management that affect the concurrent execution of rules. They include the average number of positive conditions or negative conditions in the LHS of the rules, and the average number of RHS actions (inserts and deletes) per rule. The performance measures that are significant are: the transaction throughput, the disk utilization, and the number of restarted transactions.

3.1 Closed Network Model

Figure 2 depicts the simulation closed network queue model used for the simulation. The model has two major modules, namely, the Transaction Submission Module (TSM) and the Transaction Processing Module (TSP). The TSM prepares input transactions corresponding to the qualifying rule instantiations from the CES. When a rule is committed and updates the database, the TSM will also prepare the transactions for the new rule instantiations triggered by those updates. The TSM uses the average characteristics of the rules to randomly generate sample transactions which match these characteristics. Each transaction corresponding to a rule instantiation is represented as a sequence of page requests and some corresponding rule processing instructions. These instructions correspond to reading the page, verifying negative conditions, or marking pages for deletions and insertions. They are generated in either a random fashion, or following some skew in the distribution, as required. The cost of the Match/Maintenance Process is considered an overhead within this module, since we wanted to focus on the execution of the rules.

The TPM module models the Concurrent Execution Process. Transactions are admitted for processing within TPM if and only if the number of already executing transactions in the TPM module is less than the maximum number of multiprogramming (mpl). In this way, infinite number of restarts is avoided and data and resource contention is controlled. The Multiprogramming Manager (MPL) removes transactions from the ReadyQueue and lets them start executing using the above “less than mpl” criterion.

Once the transactions pass through the MPL, they advance to the CCQueue (Concurrency Control Queue). Transactions are taken off this queue by the Concurrency Control Manager (CC). Essentially, the CC performs the scheduling of the transactions and interacts with both the lock manager and the wait-for graph. Active transactions are serviced in a round-robin preemptive fashion and the CC tries to service the next pending request of the transaction located at the top of the CCQueue. The CC interacts with the lock manager of the database to examine locking options for a particular page and proceeds accordingly. The lock table discussed earlier is implemented by the lock manager. It is assumed that the lock table is main memory resident, thus, there is virtually no overhead for lock processing.

A transaction processed by the CC may take three possible paths. The first is that the required lock is obtained and the transaction is queued at the DiskQueue for service by the Disk Manager (DM). Once the DM determines that a disk page request has been serviced, the transaction is placed in the RPRQueue to execute the rule processing instructions associated with that page. The Rule Processor (RPR) delays the transaction by the appropriate processing time associated with the corresponding instruction. It then makes a decision if the transaction must continue processing, if it is ready to commit or if it must fail.

The second path for a transaction (after CC processing) is that it is blocked due to a lock conflict. The transaction then joins the BlockQueue until this conflict ceases to exist and it will be delayed for a certain time period and then will re-enter the CCQueue for the same lock request. After a transaction spends a certain period of time in this loop without advancing to the DM, a deadlock detection algorithm is initiated by the CC. If a cycle is found in the transaction wait-for graph, a transaction is selected for abortion from the CCQueue. This is the last (third) possible path out of CC and the transaction is returned to the ReadyQueue, after it releases all its currently held locks.

The Rule Processor (RPR) decides whether a transaction must continue processing, if it is ready to commit or if it must fail. If the transaction must continue processing, it will seek further locks (page accesses) by joining the CCQueue again. A transaction fails when its condition is no longer true in the database. This occurs when the transaction reads a page where some tuples have been marked as deleted. It can also happen when new pages have been inserted into a relation so that a negative condition may be violated. When either of these situations occur, the RPR determines with a certain probability that
the rule instantiation is unsuccessful and that the transaction must be aborted. Finally, if a transaction is about to commit, it is directed to the Commit Processor (CM). The CM releases all the locks currently held by the transaction and advances the transaction back to the TSM.

### 3.2 Simulation Results

In the first set of experiments, we vary the average number of RHS actions in the rules, either inserts or deletes. We also vary the number of concurrently executing rules, \( N_c \), which is determined by the multiprogramming level (mpl). We assume a uniform lock distribution and a fairly small database with \( N_{rels} = 10 \) and \( N_{pages} = 1000 \).

In the results of Figures 3, 4, 5, we vary the average number of inserts per rule while the average number of positive and negative conditions in the LHS of the rules are fixed, as well as the average number of delete actions. Figure 3 plots the throughput for varying numbers of rules executing concurrently (varying mpl). As the average number of inserts increases (from 1 to 4), the probability that a rule has a negative condition on a relation into which a tuple(s) has already been inserted by another rule increases. This implies that the probability that such a rule is unable to verify its negative condition and is aborted is respectively higher. Note from the lock matrix that the relation level WL that are obtained to insert tuples into the relations are incompatible with the RL needed to verify the negative conditions. Consequently, as the average number of inserts per rule increases, the probability that a rule will be delayed while waiting for a relation level lock, and the probability of deadlocks and restarted transactions also increase.

Figure 3 shows that as the average number of inserts increases, and more rules are either delayed or are unsuccessfully executed, the throughput decreases. It also indicates that the throughput for a value of mpl = 5 is higher than for a value of mpl = 10. That indicates that as the number of concurrently executing rules increases, the throughput may decrease. To explain this, Figure 4 shows the number of transactions that are involved in a deadlock and are restarted. For a value of mpl = 10, the number of restarts is much higher than for a value of mpl = 5. As the average number of inserts increases, the number of restarted transactions for a value of mpl = 10 also increases more rapidly, when compared to a value of mpl = 5. This is explained by the fact that when there are more transactions executing concurrently, i.e., with higher values of mpl, and when the average number of inserts per rule also higher, then, a larger number of locks on relations are being held concurrently. This creates the number a higher number of restarted transactions due to deadlocks. In this case, the advantage of executing more rules concurrently is offset by the disadvantage of transactions restarts, and we observe a lower throughput for a value of mpl = 10.

Figure 5 plots the disk utilization as the average number of inserts per rule increases. We see that for higher
average inserts, the lower throughput (indicated in Figure 3) is accompanied by lower values of disk utilization. The figure also shows that the disk utilization for a value of \(mpl = 10\) is higher than for a value of \(mpl = 5\), although the corresponding throughput for a value of \(mpl = 10\) is actually lower (in Figure 3). The restarted transactions cause a larger number of disk accesses, thus increasing utilization. However, the throughput is measured by the number of rules that are actually executed and a larger number of restarts does not contribute to higher throughput values.

Next, we increased the average number of delete actions per rule, while keeping the positive and negative conditions as well as the insert actions fixed. Figure 6 shows the throughput versus the average number of deletes, for varying numbers of concurrently executing rules, i.e., values of \(mpl\). Figure 7 plots the number of restarted transactions versus the average number of deletes, for varying values of \(mpl\). The plots are very similar to the results obtained as we increased the average number of inserts per rule. One difference is that increasing the number of delete actions increases the number of WL obtained on pages of relations; in contrast, increasing the average number of inserts increases the number of WL at the relation level. The effect of the latter where the entire relation is locked, on the number of transactions restarted due to deadlock, is much more severe than the effect of the former. Thus, if we compare the actual number of restarted transactions in Figure 4 and Figure 7, this number is much larger in Figure 4, as we increase the average number of inserts per rule.

This also explains the difference of the serial execution \((mpl = 1)\) in Figures 3 and 6. In Figure 3, as the average number of inserts per rule increases, the throughput corresponding to the serial execution decreases. However, in Figure 6, as the average number of deletes per rule increases, the throughput increases. The cost of locking imposed by the inserts is much more severe than that imposed by the deletes (made up of the total cost of write locks on individual pages).

### 3.3 Different Locking Schemes

In this second group of experiments, we examine the effect of three different lock management methods. Figure 8 shows the throughput versus the number of concurrently executing rules for three locking schemes namely, CONV, NEW and NEW-ND. CONV is the standard 2-4 locking protocol, NEW is our protocol and NEW-ND is similar to ours except that page requests are ordered to prevent deadlocks [1]. Figure 9 shows the corresponding values for disk utilization for the three schemes. In these experiments, the average number of positive and negative conditions as well as the average number of insert and delete actions per rule are fixed, as well as the size of the database.

Figure 8 shows that for smaller \(mpl\) values NEW is superior to both NEW-ND and CONV. As the value of \(mpl\) increases, and more rules execute concurrently, the scheme NEW-ND has a better throughput than NEW, whereas the throughput for the scheme CONV is consistently lower.

For all the three schemes, the throughput increases if compared with the serial execution \((mpl = 1)\) and after reaching a maximum value it tends to decrease. As the value of \(mpl\) increases, the number of restarted transactions for CONV is much greater than the number of restarted transactions for NEW. Obviously, NEW-ND is deadlock-free so no transactions are restarted.

CONV has a greater potential for deadlocks since WLs at the relation level are incompatible with the locks at the page level. As a result, CONV has a larger number of restarted transactions compared to NEW and its throughput is correspondingly lower than NEW. We also note that in Figure 8, the throughput of the scheme CONV, for \(mpl\) values of 10 and 12 is extremely low; the simulation
data produced situations where some restarted transactions were causing livelocks, and their execution had to be terminated by the DBMS.

However, even with the new lock compatibility matrix corresponding to NEW, when more rules are executing concurrently, there are more locks held on the relations. Consequently, the number of restarted transactions increases with increasing multiprogramming values. This causes the throughput to decline for all three schemes for large values of mpl (Figure 8). If we compare NEW-ND with NEW, with smaller values of mpl, NEW has a higher throughput. This is because the NEW-ND scheme, by forcing the locks to be obtained in some particular order, delays the transactions requiring locks and there is less potential for concurrent access. Figure 9, which plots the disk utilization versus the value of mpl indicates that the percent of disk utilization for NEW-ND is always lower than that for NEW. However, larger mpl values allow for more deadlocks and the advantage of NEW over NEW-ND vanishes.

3.4 Effects of the Database Size

Figure 10, shows the throughput versus the value of mpl, for different numbers of relations. As expected, with a larger number of relation, \( N_{rels} = 40 \), the throughput increases to a much greater extent in comparison to the case where the number of relations is smaller \( N_{rels} = 10 \). In addition, with \( N_{rels} = 40 \), the throughput keeps increasing as we increase the value of mpl up to 10. In contrast, with \( N_{rels} = 10 \), the throughput reaches a maximum at mpl = 6, and then decreases. With a larger number of relations, there is less probability of deadlock and restarted transactions, for a similar number of rules executing concurrently. Hence, the disadvantage of restarted transactions are not as great, and the advantage of greater concurrent access with increasing values of mpl is obtained.

3.5 Obtaining Skewed Locks

For the final set of experiments, we study the effect of skew in obtaining locks in the database. For the relation level locks, a single hot-spot is identified as a set of one or more relations. For the page level locks, a set of pages of one or more relations is identified as a hot-spot. A skew of \( s \) percent means that \( s \) percent of the locks are distributed over the hot-spot and \( 100 - s \) percent of the locks are uniformly distributed over the entire database. In Figure 11, we show the throughput versus the percentage of locks which are skewed or directed to the hot-spot, for varying values of mpl. As the percentage of skew in the locks increases, the throughput decreases. To explain, as the percentage of skew in the locks increases, there is a higher probability of deadlocks occurring in the hot-spot. The result of this is that the number of restarted transactions increases, as seen in Figure 12. This causes the throughput to decrease as the skew increases. Figure 11 also indicates that as the percentage of skew in the locks increases, the throughput for a lower value of mpl (i.e., 5) is higher than for a higher value (i.e., 10). In order to explain, we refer to Figure 12 which indicates that the number of restarted transactions for mpl = 10 is much higher than the number of restarted transactions for mpl = 5. Due to larger value of multiprogramming, there are more transactions seeking locks in the hot-spot and the probability of deadlocks and restarted transactions, in the hot-spot, increases respectively.

4 Conclusions

The concurrent rule execution on a DBMS system was studied in this paper. Transactions were used as the encapsulation means for rules. Based on the principle of
serializability and the way conditions and actions of rules interact two locking schemes were formulated. Using a simulation testbed we experimented with these two protocols. Throughout these experiments, it is shown that rule concurrency offers significant gains when compared with the serial execution and that the extended lock table (NEW) provides greater accessibility on disk resident data over the conventional lock matrix (CONV). Updates imposed by the RHS of the rules impose considerable overhead and may degrade the performance to levels comparable to that of sequential execution. The alternative scheme NEW-ND is shown to be beneficial only in an environment where many rules run concurrently.

References


