Database Management System (DBMS)

Lecture-45

Dharmendra Kumar

January 8, 2021

Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, ..., T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and . . ., and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

There are two principal methods for dealing with the deadlock problem. We can use a deadlock prevention protocol to ensure that the system will never enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a deadlock detection and deadlock recovery scheme. **Note:** Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

Deadlock Prevention

Two different deadlock prevention schemes using timestamps have been proposed:

1. wait–die: This scheme is a non-preemptive technique. When transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp smaller than that of T_j (that is, T_i is older than T_j). Otherwise, T_i is rolled back (dies).

For example, suppose that transactions T_1, T_2, and T_3 have timestamps 5, 10, and 15, respectively. If T_1 requests a data itemheld by T_2, then T_1 will wait. If T_3 requests a data item held by T_2, then T_3 will be rolled back.

2. wound–wait: This scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp larger than that of T_j (that is, T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is wounded by T_i).

Returning to our example, with transactions T_1, T_2, and T_3, if T_1 requests a data item held by T_2, then the data item will be preempted from T_2, and T_2 will be rolled back. If T_3 requests a data item held by T_2, then T_3 will wait.

Deadlock Detection and Recovery

If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock.

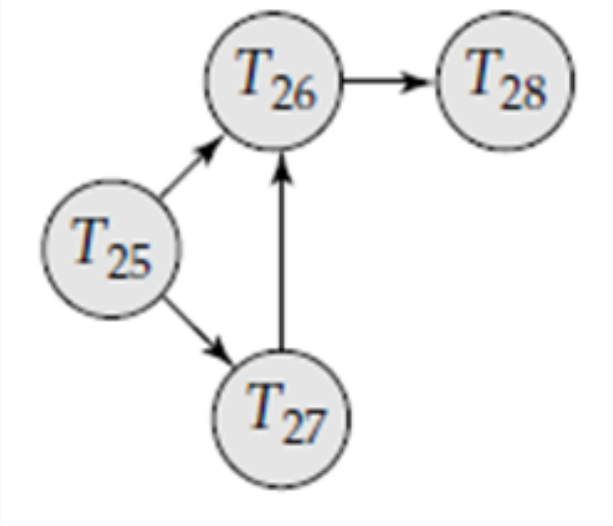
Deadlock Detection

To identify the deadlock is present in the system, we use a directed graph called **wait-for-graph**.

In this graph, vertices are corresponding to transactions. When transaction T_i requests a data item currently being held by transaction T_j , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction $T_j$is no longer holding a data item needed by transaction T_i.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

Example: Consider the wait-for graph show in the following figure,

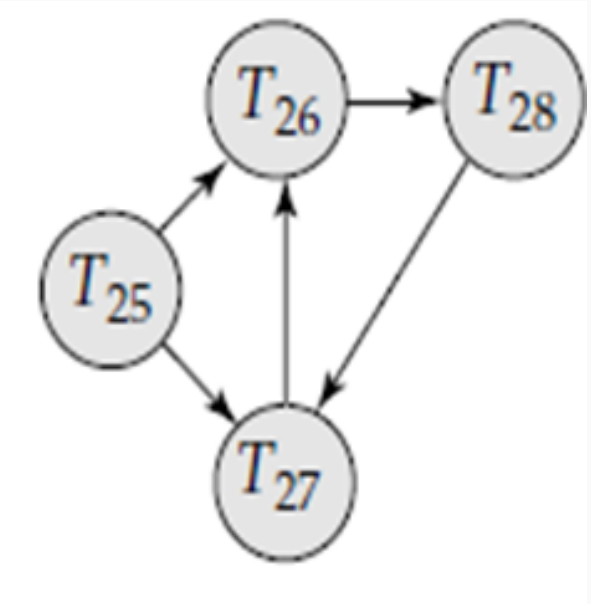


which depicts the following situation:

- Transaction T_{25} is waiting for transactions T_{26} and T_{27}.
- Transaction T_{27} is waiting for transaction T_{26}.
- Transaction T_{26} is waiting for transaction T_{28}.

Since the graph has no cycle, the system is not in a deadlock state

Suppose now that transaction T_{28} is requesting an item held by T_{27}. The edge $T_{28} \rightarrow T_{27}$is added to the wait-for graph, resulting in the new system state in following figure.



This time, the graph contains the cycle
$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$.
implying that transactions T_{26}, T_{27}, and T_{28} are all deadlocked.

Concurrency Control

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. Selection of a victim: Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock.We should roll back those transactions that will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may determine the cost of a rollback, including

1. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
2. How many data items the transaction has used.
3. How many more data items the transaction needs for it to complete.
4. How many transactions will be involved in the rollback.

2. Rollback: Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
The simplest solution is a total rollback:

3. Starvation: In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is starvation. We must ensure that transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

The Phantom Phenomenon

Consider transaction T_{29} that executes the following SQL query on the bank database:

```
select sum(balance)
from account
where branch-name = 'Perryridge'
```

Transaction T_{29} requires access to all tuples of the account relation pertaining to the Perryridge branch.

Let T_{30} be a transaction that executes the following SQL insertion:

```
insert into account
values (A-201, 'Perryridge', 900)
```

Concurrency Control

Let S be a schedule involving T_{29} and T_{30} . We expect there to be potential for a conflict for the following reason:

- If T_{29} uses the tuple newly inserted by T_{30} in computing sum(balance), then T_{29} read a value written by T_{30}. Thus, in a serial schedule equivalent to S, T_{30} must come before T_{29}.
- If T_{29} does not use the tuple newly inserted by T_{30} in computing sum(balance), then in a serial schedule equivalent to S, T_{29} must come before T_{30}.

The second of these two cases is curious. T_{29} and T_{29} do not access any tuple in common, yet they conflict with each other! In effect, T_{29} and T_{29} conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. This problem is called the **phantom phenomenon**. To prevent the phantom phenomenon, we allow T_{29} to prevent other transactions from creating new tuples in the account relation with branch-name = "Perryridge."