Homework 3 Distributed Database Systems

Christoffer Brevik

November 18, 2024

1 Find conflict equivalent schedules

We have the following schedules:

1. S1 = W2(x), W1(x), R3(x), R1(x), C1, W2(y), R3(y), R3(z), C3, R2(z), C2
2. S2 = R3(z), R3(y), W2(y), R2(z), W1(x), R3(x), W2(x), R1(x), C1, C2, C3
3. S3 = R3(z), W2(x), W2(y), R1(x), R3(x), R2(z), R3(y), C3, W1(x), C2, C1
4. S4 = R2(z), W2(x), W2(y), C2, W1(x), R1(x), A1, R3(x), R3(z), R3(y), C3

We are tasked with classifying which schedules are serializable, and which are conflict equivalent.

1.1 Histories

From the schedules we can make the following histories, to better visualize when which operation is completed. I've removed the commit operatons as these are not relevant:

Schedule S1

Time	$T1$	$T2$	$T3$
(1)		$W_2(x)$	
(2)	$W_1(x)$		
(3)			$R_3(x)$
(4)	$R_1(x)$		
(5)		$W_2(y)$	
(6)			$R_3(y)$
(7)			$R_3(z)$
(8)		$R_2(z)$	

Schedule S2

Time	$T1$	$T2$	$T3$
(1)			$R_3(z)$
(2)			$R_3(y)$
(3)		$W_2(y)$	
(4)		$R_2(z)$	
(5)	$W_1(x)$		
(6)			$R_3(x)$
(7)		$W_2(x)$	
(8)	$R_1(x)$		

Schedule S3

Time	$T1$	$T2$	$T3$
(1)			$R_3(z)$
(2)		$W_2(x)$	
(3)		$W_2(y)$	
(4)	$R_1(x)$		
(5)			$R_3(x)$
(6)		$R_2(z)$	
(7)			$R_3(y)$
(8)	$W_1(x)$		

Schedule S4

Time	$T1$	$T2$	$T3$
(1)		$R_2(z)$	
(2)		$W_2(x)$	
(3)		$W_2(y)$	
(4)	$W_1(x)$		
(5)	$R_1(x)$		
(6)			$R_3(x)$
(7)			$R_3(z)$
(8)			$R_3(y)$

Table 1: Conflicting operations in S1, S2, S3, and S4, ignoring Commits and Abort operations

1.2 Conflict Pairs

S1

In schedule $S1$, the conflicting operations are as follows:

- **Conflicts on x:**
  - $W_2(x) \rightarrow W_1(x)$
  - $W_2(x) \rightarrow R_3(x)$
  - $W_2(x) \rightarrow R_1(x)$
  - $W_1(x) \rightarrow R_3(x)$

  - $W_1(x) \rightarrow R_1(x)$

- **Conflicts on y:**
  - $W_2(y) \rightarrow R_3(y)$
- **Conflicts on z:**
  - No Conflicts

S2

In schedule $S2$, the conflicting operations are as follows:

- **Conflicts on x:**
  - $W_1(x) \rightarrow R_3(x)$
  - $W_1(x) \rightarrow R_1(x)$
  - $R_3(x) \rightarrow W_2(x)$
  - $W_2(x) \rightarrow R_1(x)$
- **Conflicts on y:**
  - $R_3(y) \rightarrow W_2(y)$
- **Conflicts on z:**
  - No Conflicts

S3

In schedule $S3$, the conflicting operations are as follows:

- **Conflicts on x:**
  - $W_2(x) \rightarrow R_1(x)$
  - $W_2(x) \rightarrow R_3(x)$
  - $W_2(x) \rightarrow W_1(x)$
  - $R_1(x) \rightarrow W_1(x)$
  - $R_3(x) \rightarrow W_1(x)$
- **Conflicts on y:**
  - $W_2(y) \rightarrow R_3(y)$
- **Conflicts on z:**
  - No Conflicts

S4

In schedule $S4$, the conflicting operations are as follows:

- **Conflicts on x:**
  - $W_2(x) \rightarrow W_1(x)$
  - $W_2(x) \rightarrow R_1(x)$
  - $W_2(x) \rightarrow R_3(x)$
  - $W_1(x) \rightarrow R_1(x)$
  - $W_1(x) \rightarrow R_3(x)$

- **Conflicts on y:**
  - $W_2(y) \rightarrow R_3(y)$
- **Conflicts on z:**
  - No Conflicts

1.3 Conflict Equivalence

To check for conflict equivalence among the schedules, we compare the relative ordering of conflicting operations in each schedule. As most operations affect x, we can first look at y and z.

Firstly we see that no schedules have any conflict in z, but that all sharte the same $W_2(y) \rightarrow R_3(y)$, except **S2**. This means that we can immediatly say that **S2 is not conflict equivalent with any of the other schedules**. For the other three schedules, we must compare the conflicting operations affecting x:

- **Comparing S2 and S1:** Established non-equivalence
- **Comparing S2 and S3:** Established non-equivalence
- **Comparing S2 and S4:** Established non-equivalence
- **Comparing S1 and S3:**
  - In **S3** we see that $W_1(x)$ comes after both $R_1(x)$ and $R_3(x)$, where as in **S1** it comes beforehand
  - The schedules are therefore not conflict equivalent
- **Comparing S1 and S4:**
  - Both Writes to x are before any reads
  - $W_2(x)$ comes before $W_1(x)$ in both schedules
  - The read order is irellevant
  - All conflicting operations are in the same order. S1 and S4 are conflict equivalent
- **Comparing S3 and S4:**
  - In **S3** we see that $W_1(x)$ comes after both $R_1(x)$ and $R_3(x)$, where as in **S4** it comes beforehand
  - The schedules are therefore not conflict equivalent

1.4 Serializable Schedules

To determine whether each schedule is serializable, we check the conflicting operations and see if there is any cycles for the dependencies. As we have only one conflicting operations for y and none for z, we only look at conflicting operations for x based on the tables in 1.1 and the list in 1.2:

1.4.1 S1

We see that T2 operaitons are not preceeded by T1 or T3 conflicting operations, and that T1 is only preceeded by T2. Therefore there are no cycling dependencies and **S1 is serializable**.

1.4.2 S2

We see that T2 both have conflicting operations preceding ($W_2(y)$) and following ($W_2(x)$) operations in T1. Therefore we have a cycling dependency and **S2 is not serializable**.

1.4.3 S3

We see that T2 operaitons are not preceeded by conflicting operations from T1 or T3, and that T1 is only preceeded by T2. Therefore there are no cycling dependencies and **S3 is serializable**.

1.4.4 S4

Here all operations are used according to the order T2, T1, T3, with only operations in T2 starting after T1 is done. Therefore there can't be any cycling dependencies and **S4 is serializable**

1.5 Conclustion

We conclude that from the 4 schedules presented, S1 and S4 are conflict equivalent. And that S1, S3 and S4 are serializable as they contain no cycling dependencies.

2 Comparison of Centralized and Hierarchical Deadlock Detection Approaches

In this section, I will discuss the centralized and hierarchical deadlock detection approaches, as described in the course material, and compare their respective advantages and disadvantages in the context of distributed database management systems (DBMS).

2.1 Structure and Organization

Centralized Deadlock Detection

In centralized deadlock detection, a single site, known as the *central detector*, is responsible for managing deadlock handling across the entire system. At regular intervals, all lock managers in the system report their Local Wait-For Graph (LWFG), which details the wait-for relations among transactions, to the central detector. The detector then combines all LWFGs to create a Global Wait-For Graph (GWFG). By analyzing the GWFG, it can identify cycles indicating that processes are mutually blocking each other, thus causing deadlocks. In practice, lock managers typically only send changes made since the last reporting interval, rather than the entire LWFG, to reduce communication overhead.

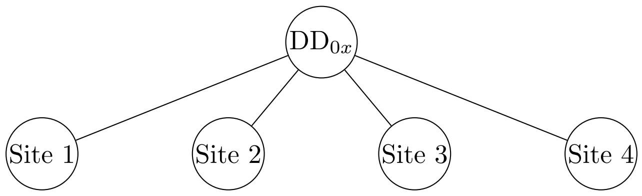

Figure 1: Centralized Deadlock Detection

This approach is straightforward to implement and maintain, as the responsibility of deadlock detection is centralized in one location. However, this simplicity comes with potential drawbacks. The reliance on a single central detector creates a communication bottleneck, particularly as the system scales, since all lock managers must communicate with the same site. Additionally, this structure introduces a single point of failure; if the central detector fails, deadlock detection is halted across the entire system.

Hierarchical Deadlock Detection

Hierarchical deadlock detection distributes the responsibility of deadlock detection across multiple levels within a tree-like hierarchical structure, addressing some of the limitations of centralized detection by creating a hierarchy of deadlock detectors (DDs) across different sites.

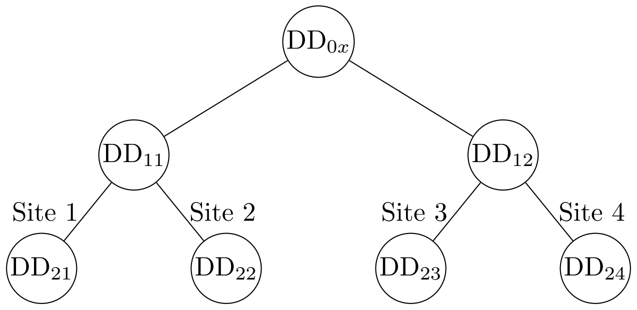

Figure 2: Example Implementation of Hierarchical Deadlock Detection

In this structure, each site has a local deadlock detector that identifies deadlocks confined to its own transactions using an LWFG. Deadlocks affecting only a single site are detected locally (e.g., DD_{21} at Site 1 in Figure 2). When potential deadlocks span multiple sites, each site's LWFG is sent to a higher-level detector. These higher-level detectors analyze the combined LWFGs from multiple sites under their control, enabling them to detect distributed deadlocks that affect two or more sites.

For example, if a deadlock involves transactions at both Site 1 and Site 2, it is detected by the level-1 detector DD_{11}, which oversees both sites. If the deadlock spans across sites in different branches of the hierarchy (e.g., Site 1 and Site 4), it will be detected by the top-level detector DD_{0x}, which monitors the entire system.

This hierarchical approach reduces communication costs by localizing deadlock detection to relevant parts of the hierarchy and reducing dependence on a single central site. However, it is complex to implement, requiring sophisticated coordination among the different levels of detectors. This complexity often necessitates non-trivial modifications to the lock and transaction manager algorithms to ensure effective deadlock management across the hierarchy.

2.2 Comparing the two approaches

In summary, centralized deadlock detection is a straightforward approach that may be effective for smaller, less complex systems where communication and computational resources are not heavily constrained. Its simplicity makes it easier to implement and maintain but limits its scalability due to the central bottleneck and vulnerability to a single point of failure.

Hierarchical deadlock detection, on the other hand, offers a more scalable solution suited to large, distributed DBMS environments. By distributing responsibilities across multiple levels, it reduces communication overhead and improves fault tolerance, as local issues can be handled at lower levels without requiring immediate central intervention. However, the added complexity in setup and management may pose challenges, particularly in systems that lack the infrastructure or coordination mechanisms required for hierarchical control. Overall, the choice between these two approaches depends on the specific needs and scale of the DBMS environment in question. Below is a table comparing the two approaches:

2.3 Summary Table

Feature	**Centralized Deadlock Detection**	**Hierarchical Deadlock Detection**
Structure	Single central controller	Multi-level hierarchy
Communication Overhead	High (single-point communication)	Moderate to low (localized communication)
Scalability	Limited by central bottleneck	High, more scalable
Fault Tolerance	Low, single point of failure	High, distributed responsibilities
Detection Delay	Low in small systems, high in large systems	Moderate, but generally efficient for large systems
Implementation	Simple	Complex

Table 2: Comparison of Centralized and Hierarchical Deadlock Detection Approaches

3 Analysis of Modified Local Wait-For Graph with $T_{\mathbf{ex}}$

Consider a modification to the local wait-for graph by introducing a new node T_{ex}. For every transaction T_i that is waiting for a lock at an external site, an edge $T_i \to T_{\text{ex}}$ is added. Similarly, if an external transaction is waiting for T_i to release a lock at this site, an edge $T_{\text{ex}} \to T_i$ is added.

We can make the following conclusions based on the presence of cycles in this modified wait-for graph:

1. **Cycle Not Involving $T_{\mathbf{ex}}$:**
   - If there is a cycle in the modified local wait-for graph that does **not** involve T_{ex}, this means that there is a deadlock among the local transactions themselves, independent of any external transactions.
   - Since T_{ex} is not part of the cycle, all dependencies are internal to the current site. Thus, we can conclude that there is a **local deadlock** among the transactions at this site.
2. **Cycle Involving $T_{\mathbf{ex}}$:**
   - If there is a cycle in the modified local wait-for graph that **involves** T_{ex}, this indicates that the deadlock is not confined to local transactions alone. Instead, it involves a dependency on an external transaction.
   - In this case, we can conclude that there is a **distributed deadlock** involving transactions across multiple sites. The presence of T_{ex} in the cycle implies that at least one transaction at this site is waiting for a lock held by an external transaction, or vice versa, creating a cross-site dependency loop.

4 Analysis of Global Deadlock Detection Scheme

This scheme, in which each site combines its local waits-for graph (LWFG) with the cumulative graph received from the previous site before forwarding it, is generally effective for detecting global deadlocks. As each site receives the cumulative waits-for graph, it adds its own local dependencies and checks for cycles indicating potential deadlocks. By the time the graph completes a full loop back to the originating site, it should contain the combined dependencies across all sites, revealing any cycles that signify a global deadlock. However, this approach is not flawless, and there are scenarios where it may fail to construct an accurate Global Waits-For Graph (GWFG).

In distributed systems, network delays can cause the cumulative waits-for graph to become outdated by the time it reaches the original site. Additionally, network partitioning could prevent certain sites from contributing their dependency information, potentially leading to missed cycles if the deadlock involves nodes across partitioned sites.

These challenges are also prevalent in both centralized and hierarchical deadlock detection approaches. The sequential nature of this solution, where each site analyzes and sends updated information to the next, introduces significant communication overhead, which further increases the likelihood of delays and outdated information.

Therefore, while this solution is simple, it is not ideal for accurately detecting global deadlocks in a distributed environment due to these limitations, and running the algorithm **does not guarantee to detect a deadlock in the system**.

If we are to assume a perfect environment, where network delays aren't a factor to worry about, then I do not find any issues with the solution. Therefore if this is the case I don't see why it would be able to guarantee that a deadlock would be found.