# Control de la concurrencia

CI-0127 Bases de Datos, Universidad de Costa Rica

Sivana Hamer

## Importante

Este documento recopila contenidos de diversos de sitios web especializados, académicos y documentos compartidos por universidades. Toda la información es utilizada con fines estrictamente académicos. Para más información, puede ver las referencias.

## 1 Introduction

Usually, transaction processing systems allow multiple transactions to run concurrently causing inconsistencies with the data. Though running transaction *serially* (one at a time) is easier, there are benefits for using concurrency:

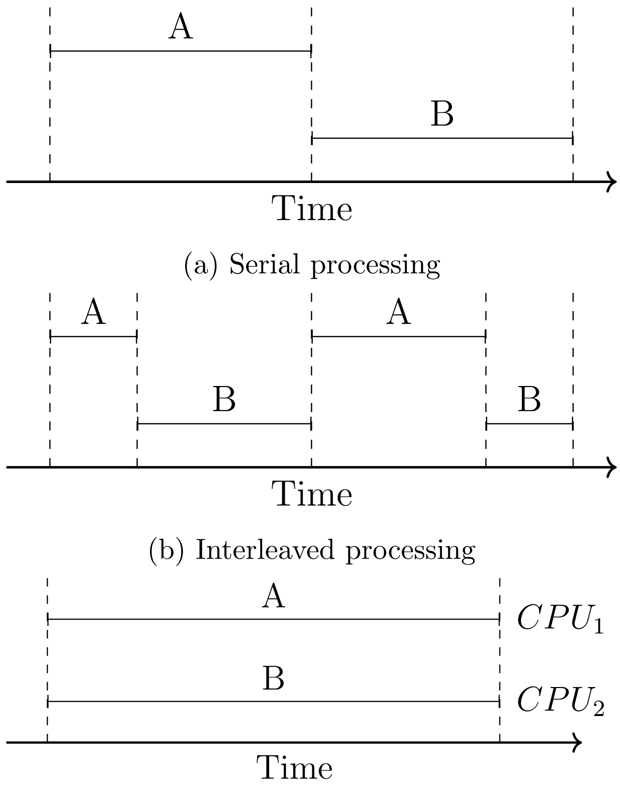
- **Improved throughput and resource utilization.** As there are many steps in a transaction, some involving the CPU and others the I/O, both can be executed in parallel. One can have one transaction reading from a disk and another using the CPU. If there are multiple disks, another transaction can also read from it. Therefore the number of transactions executed at a time (*throughput*), and idle time for CPU and disks reduce (*resource utilization*).
- **Reduced waiting time.** The running time of a transaction may vary, thus if transactions run serially a short transition might have unpredictable delays waiting for a longer transaction to finish. Running a transaction concurrently, if they use different parts for the database, is better as it allows them to share resources. Thus it reduces unpredictable delays, bringing down the average time that a transaction takes to be completed after submission (*average response time*).

Still, as concurrency allows for multiple transactions to be run at the same time we must ensure the *isolation* ACID property. To do this, we must use *schedules* to identify the order of execution (Sections 3). Furthermore to ensure the consistency of the database we use several mechanisms called *concurrency-control protocols* (Section 6).

# 2 Concurrency

While some systems may have a database system with a single-user, it is common for database systems to be *multi-user*. A simple way we could handle multiple transactions is executing one after the other with only one thread (Subfig. 1a). However, this is inefficient in time as there is only one transaction executing and we require to copy the database per transaction.

Another possible approach is to process the transaction *concurrently* in a thread using *interleaved processing* by partially executing the transaction and then suspending it to execute other operations (Subfig. 1b). Furthermore, we could also use multiple threads to allow for *parallel processing* (Subfig. 1c).




(a) Serial processing

(b) Interleaved processing

(c) Parallel processing

Figure 1: Execution of processes $A$ and $B$

# 3 Schedules

The chronological order of execution of sequences of $n$ transactions $T_1, T_2, \cdots, T_n$ is the *schedule* or *history*. The order of the instructions inside the transaction must be preserved.

Suppose we have a transaction $T_1$ that transfers \$100 from Alice's bank account ($A$) to Bob's bank account ($B$) shown in Fig. 2. We also have a transaction $T_2$ in which \$50 are deposited in cash to Alice's bank account ($A$) shown in Fig. 3. Therefore a schedule would be the order of execution for both of these transactions. At the end of each transaction to detail that the transaction has entered a committed state, we use the *commit* instruction.

$$
\begin{array}{ll}
T_1: & \text{read}(A); \\
 & A := A - 100; \\
 & \text{write}(A); \\
 & \text{read}(B); \\
 & B := B + 100; \\
 & \text{write}(B);
\end{array}
$$

Figure 2: Transfering \$100 from Alice's bank account ($A$) to Bob's bank account ($B$)

$$
\begin{array}{ll}
T_2: & \text{read}(A); \\
 & A := A + 50; \\
 & \text{write}(A);
\end{array}
$$

Figure 3: Cash deposit of \$50 to Alice's bank account ($A$)

An example of a *serial* schedule is shown in Fig. 4. The instructors are shown in chronological order from top to bottom, with the instructions for $T_1$ and $T_2$ shown in the left and right column, respectively. If there was \$1000 in $A$ and \$500 in $B$, the result of the scheduled execution is \$950 in $A$ and \$600 in $B$. As the total money in the accounts is equal to $A + B + \$50$, the data is consistent. Furthermore, if $T_2$ and then $T_1$ is executed, shown in Fig. 5, the result will also be consistent.

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| write($A$); | |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |

Figure 4: Schedule 1. Serial schedule of $T_1$ after $T_2$

We could also execute transactions *concurrently*, generating many possible sequences as operations that can be *interleaved*. However, we cannot predict how many instructions will be executed by the CPU before switching to another transaction.

| $T_1$ | $T_2$ |
|---|---|
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| read($A$); | |
| $A := A - 100$; | |
| write($A$); | |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |

Figure 5: Schedule 2. Serial schedule of $T_2$ after $T_1$

A posible schedule is shown in Fig. 6, where the total money in the accounts after the transaction is $A + B + \$50$. Thus this schedule was equivalent to one that was executed serially. However, not all concurrent executions will result in a consistent state. The schedule shown in Fig. 7 after execution will result with \$900 in $A$ and \$600 in $B$. Therefore, it is an inconsistent state as the \$50 deposit is lost. Therefore, we cannot leave to the operating system the control of possible schedules as it may result in inconsistent states. The database system will be in a chart of executing schedules that result in consistent states using the *concurrency-control* component. Therefore, to ensure the consistency a concurrent schedule must be equivalent to a serial schedule. Such schedules are *serializable* (Section 5).

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| write($A$); | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |

Figure 6: Schedule 3. Concurrent schedule equivalent to serial execution

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| write($A$); | |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |

Figure 7: Schedule 4. Concurrent schedule resulting in an inconsistent state

# 4 Concurrency-control problems

There are many problems that can happen while we execute our queries concurrently.

## Lost update

A *lost update* (dirty write) occurs when two transactions interleave in such a way that the value produced by the database is incorrect. $W_i$ represents a write to transaction $T_i$ and a $R_i$ a read to transaction $T_i$. The anomaly occurs for $T_i$ when the schedule interleaves for $Q$ in such a way where $W_i(Q), \cdots, W_j(Q)$.

An example is shown in Fig. 8. When $T_2$ writes the result of $A$ it is incorrect as it reads the value of $A$ before $T_1$ changes the database result. Thus, the result in $A$ will be \$1050 instead of \$950 due to losing the debit of \$100 to the bank account.

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| | read($A$); |
| | $A := A + 50$; |
| **write($A$);** | |
| | **write($A$);** |
| read($B$); | |
| $A := B + 100$; | |
| write($B$); | |

Figure 8: Lost update example

## Dirty read

A *dirty read* (temporary update) occurs when a transaction updates a value that then fails, while another transaction reads the data before the roll back. The problem occurs for $T_i$ when the schedule interleaves for $Q$ in such a way where $W_j(Q), \cdots, R_i(Q)$.

An example is shown in Fig. 9. $T_2$ reads the temporary result of $A$. However, $T_1$ has a failure and the previous data is recovered to the original value. Therefore, $T_2$ reads *dirty data* created by the transaction that has not been completed and committed.

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| **write($A$);** | |
| | **read($A$);** |
| | $A := A + 50$; |
| | write($A$); |
| read($B$); | |
| ABORT; | |

Figure 9: Dirty read example

## Unrepeatable read

An *unrepeatable read* (fuzzy or non-repeatable read) occurs when a transaction reads the same data twice, but the value was changed by another transaction between reads. The issue occurs for $T_i$ when the schedule interleaves for $Q$ in such a way where $R_i(Q), \cdots, W_j(Q), \cdots R_i(Q)$.

An example is shown in Fig. 10. $T_1$ reads a result for $A$ at the beginning of the transaction that is modified by $T_2$. When $T_1$ reads the data of $A$ again there are *different values* for the same data.

| $T_1$ | $T_2$ |
|---|---|
| **read($A$);** | |
| $A := A - 50$; | |
| write($A$); | |
| | read($A$); |
| | $A := A + 50$; |
| | **write($A$);** |
| **read($A$);** | |
| $A := A - 50$; | |
| write($A$); | |

Figure 10: Unrepeatable read example

## Phantom read

A phantom read occurs when a transaction repeats a *search condition* but gets a different a *set of items* that satisfies the condition. [$y$ in $Q$] represents modifying (inserting, updating or deleting) a tuple $y$ for the data item $Q$ . The anomaly occurs for $T_i$ when the schedule interleaves for $Q$ in such a way where $R_i(Q), \cdots, W_j[y \text{ in } Q], \cdots R_i(Q)$.

An example is shown in Fig. 11. $T_1$ reads the total of $tA$ money transferred in Alice's bank account, However, in $T_2$ a new transfer is added to all the money transfers $tA$. Therefore, when the same condition is executed in $T_1$ a new *phantom tuple* exists that was not previously there.

| $T_1$ | $T_2$ |
|---|---|
| **read($tA$);** | |
| | **insert($a_{n+1}$ in $tA$);** |
| **read($tA$);** | |

Figure 11: Phantom read example

# 5 Serializability

A *serializable* schedule determines if the order of concurrent operations are equivalent to the serial execution. The *serializability* identifies when a schedule is serializable. Serial schedules are serializable, and interleaved executions can also be serializable though it is harder to determine. There are two types of serializability: conflict and view. Neither definition encapsulates all serializable schedules.

## Conflict serializability

If we have a schedule $S$ with two consecutive instructions $I$ and $J$ of transactions $T_i$ and $T_j$, with $i \neq j$. If $I$ and $J$ are different data items the steps can be swapped without affecting the results. However if $I$ and $J$ both refer to the same data item $Q$ then the order may matter. As in our transactions we only have *read* and *write* instructions only four possibilities can happen:

- **read-read:** $I = read(Q), J = read(Q)$. The order does not matter as, regardless of the order of $I$ and $J$, the value $Q$ will be the same.
- **read-write:** $I = read(Q), J = write(Q)$. The order matters. If $T_i$ first reads with instruction $I$ $Q$ and then $T_j$ writes to $Q$ a value, then $T_i$ will not have the value written by $T_j$. However, if $T_j$ writes first a value in $Q$ in instruction $T$, then $T_i$ will read in $I$ the updated $Q$ value.
- **write-read:** $I = write(Q), J = read(Q)$. The order does matter, for the same reason as the previous case.
- **write-write:** $I = write(Q), J = write(Q)$. The order does not affect $T_i$ and $T_j$. However, for the next $read(Q)$ operation the order matters as only the last *write* instruction is preserved in the database. If there is no other $write(Q)$ the order does affect the final value of $Q$ in the database.

Therefore, if either the $I$ or $J$ instruction performs a *write* operation on the same data item there is a *conflict*. For example, the schedule 3 shown in Fig. 6 has a conflict between the $write(A)$ of $T_1$ with $read(A)$ of $T_2$. However, $write(A)$ of $T_2$ has no conflict with $read(B)$ of $T_1$ as they access different data items.

If $I$ and $J$ are two consecutive instructions of a schedule $S$ that do not have a conflict, then we can swap the order of $I$ and $J$ to generate a new schedule for $S'$. $S$ is equivalent to $S'$ as all instructions have the same order except for $I$ and $J$ whose order does not matter. If schedule $S$ can be transformed into a schedule $S'$ by swapping non-conflicting instructions, then $S$ and $S'$ are *conflict equivalent*.

Fig. 12 shows the swaps to transform the schedule 3 (Fig. 6) to the serial schedule 1 (Fig. 4). We only swap the *reads* and *writes* as these are the only operations considered.

A schedule $S$ is *conflict serializable* if it is conflict equivalent to a serial schedule. For example, schedule 3 is conflict serializable to schedule 1. However, schedule 4 (Fig. 7) is not conflict serializable as it is not equivalent to either schedule 1 or schedule 2.

Another way to determine if the conflict serializability of a schedule is with a *dependency graph* or *precedence graph*. We create a node for each of the following operations. The graph $G$ will have pairs of $G = (V, E)$. The set of vertices $V$ are all the transactions in the schedule. The edges $E$ are created if for $T_i$ to $T_j$ if any of the following conditions (conflicts) hold:

- $T_i$ executes a $write(Q)$ before $T_j$ executes a $read(Q)$.
- $T_i$ executes a $read(Q)$ before $T_j$ executes a $write(Q)$.
- $T_i$ executes a $write(Q)$ before $T_j$ executes a $write(Q)$.

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| | write($A$); |
| read($B$); | |
| write($B$); | |

(a) Concurrent schedule

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| **read($B$);** | |
| | **write($A$);** |
| write($B$); | |

(b) $T_1$ *read*($B$) $\leftrightarrow$ $T_2$ *write*($A$)

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| read($B$); | |
| **write($B$);** | |
| | **write($A$);** |

(c) $T_1$ *write*($B$) $\leftrightarrow$ $T_2$ *write*($A$)

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| **read($B$);** | |
| | **read($A$);** |
| write($B$); | |
| | write($A$); |

(d) $T_1$ *read*($B$) $\leftrightarrow$ $T_2$ *read*($A$)

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| read($B$); | |
| **write($B$);** | |
| | **read($A$);** |
| | write($A$); |

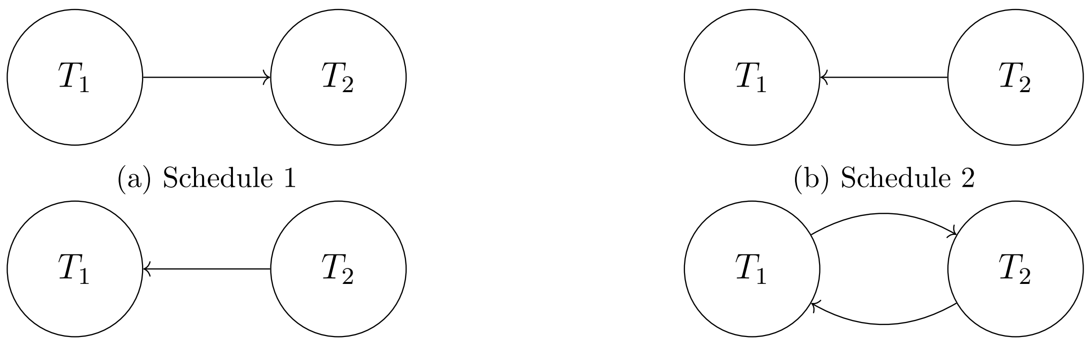
(e) $T_1$ *read*($B$) $\leftrightarrow$ $T_2$ *read*($A$)

Figure 12: Transforming a concurrent schedule to an equivalent serial schedule

If there is an edge $T_i$ to $T_j$ exists in the graph, then in a serial execution $S'$ $T_i$ must execute before $T_j$. If there are no cycles, then the schedule $S$ is conflict serializable. If the graph has a cycle, then $S$ is not conflict serializable.

For example, the precedence graphs of schedules 1 to 4 are shown in Fig. 13.

- For schedule 1 (Subfig. 13a) there is a single edge $T_1 \rightarrow T_2$ as $T_1$ executes *write*($A$) before $T_2$ executes a *read*($A$).
- Schedule 2 (Subfig. 13b) has also only one edge $T_2 \rightarrow T_1$ for a similar reason than Schedule 1. The edge from $T_2 \rightarrow T_1$ exists as $T_2$ executes *write*($A$) before $T_2$ executes a *read*($A$).
- Schedule 3 (Subfig. 13c) has one edge $T_1 \rightarrow T_2$ as $T_1$ executes *write*($A$) before $T_2$ executes a *read*($A$). Therefore, the concurrent schedule 3 is conflict serializable.
- For schedule 4 (Subfig. 13d), there is an edge $T_1 \rightarrow T_2$ as $T_1$ executes *read*($A$) before $T_2$ executes a *write*($A$). There is also an edge $T_2 \rightarrow T_1$ as $T_2$ executes *write*($A$) before $T_2$ executes *write*($A$). Therefore, there is a cycle and the schedule is not serializable.




(a) Schedule 1 (b) Schedule 2 (c) Schedule 3 (d) Schedule 4

Figure 13: Precedence graph of schedules 1 through 4

The *serializability order* defines the order of execution of the transactions that is consistent with the partial order of the precedence graph. To test for conflict serializability, the precedence graphs is constructed and a cycle-detection algorithm used.

## View serializability

If there is a transaction $T_3$, shown in Fig. 14, that transfers \$20 from Bob's bank account $(B)$ to Alice's bank account $(A)$. A schedule 5 could be created by executing $T_1$ and $T_3$, as shown in shown in Fig. 15. The precedence graph of schedule 5 has an edge from $T_1 \rightarrow T_3$ as $T_1$ executes $write(A)$ before $T_3$ executes $read(A)$, and an edge from $T_3 \rightarrow T_1$ as $T_3$ executes $write(B)$ before $T_1$ executes $read(B)$. Thus, there is a cycle and the transaction is not conflict serializable.

$T_3$ : read($B$);
$B := B - 20$;
write($B$);
read($A$);
$A := A + 20$;
write($A$);

Figure 14: Transfering \$20 from Bob's bank account $(A)$ to Alice's bank account $(B)$

| $T_1$ | $T_3$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| write($A$); | |
| | read($B$); |
| | $B := B - 20$; |
| | write($B$); |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| | read($A$); |
| | $A := A + 20$; |
| | write($A$); |

Figure 15: Schedule 5. Concurrent view serializable schedule of $T_1$ and $T_3$

However, if we execute the transaction we will get an equivalent result to a serial schedule of $< T_1, T_3 >$ due to the fact that the mathematical increment and decrement operations are commutative. Therefore, there are schedules that produce the same outcome but are not conflict equivalent. Analyzing the results instead of only considering *read* and *write* operations is the *view serializability*. However, this type of serializability is not used in practice because it is computationally complex to determine (NP-complete).

# 6 Concurrency-control protocols

To ensure the *isolation* ACID property, there are several possible mechanisms or techniques known as *concurrency-control protocols* or *concurrency-control schemes*. These protocols ensure the proper execution of transactions when they are concurrent and interleaved. Thus, they generate an execution schedule equivalent to a serial schedule. Furthermore, the protocols cannot know if there are conflicts ahead of time.

There are two main categories for the protocols:

- **Pessimistic:** The DBMS assumes that transactions *will have conflicts*, therefore it doesn't allow the problems to occur. Lock-based (Section 7) are pessimistic.
- **Optimistic:** The DBMS assumes that *conflicts between transactions are rare*, therefore it assumes it will be able to finish the transaction after commiting. Checks are performed after the transaction is executed. Validation-based protocols (Section 11) and timestamp-based protocols (Section 10) are optimistic.

# 7 Lock-based protocols

To ensure that only one transaction is modifying a data item at a time (*mutually exclusive manner*) a *lock* can be used on the data item. The basic types of locks are:

- **Shared lock ($S - LOCK$).** Several transactions can read at the same time, but none can write. This lock can be acquired by multiple transactions at the same time.
- **Exclusive lock ($X - LOCK$).** Only one transaction can both read and write. This lock prevents other transactions acquiring $S - LOCK$ or $X - LOCK$.

We request the $S - LOCK$ for a data item $Q$ executing the **S-LOCK(Q)** instruction, while for $X - LOCK$ we execute the **X-LOCK(Q)** instruction. To release either lock on a data item $Q$, we execute the **UNLOCK(Q)** instruction. If we hold the lock till the end of a transaction (after a commit or abort) it is a *long lock*, while if we liberate it before it is a *short lock*.

The transactions *request* locks (or upgrades) to the *concurrency-control manager*. The lock requested depends on the type of transaction performed. The transaction will have to *wait* to continue it's execution until the concurrency-control manager *grants* the lock. The lock will be granted until when all the incompatible locks held by other transactions have been released. When a transaction finishes using a lock, it must be *released* so other transactions can use it.

*Compatible* modes do not require waiting to be granted permission to use the lock if another transaction currently holds a lock. The compatibility between the modes is shown in Fig. 16. Only $S - LOCK$s are compatible with each other and can be held simultaneously.

| | $S - LOCK$ | $X - LOCK$ |
|---|---|---|
| $S - LOCK$ | ✓ | ✗ |
| $X - LOCK$ | ✗ | ✗ |

Figure 16: Compatibility matrix of $S - LOCK$ and $X - LOCK$

Locks can have different types of issues:

- If we release a lock too soon after reading or writing, an *inconsistent state* may occur. For example, schedule 6 shown in Fig. 17 there is an inconsistent state for $T_1$ as it was modified by $T_2$ during its execution. Therefore, locks by themselves do not ensure serializability. Inconsistent states can cause real-world problems, thus these cannot be tolerated by the DBMS.

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| read(A); | |
| write(A); | |
| **UNLOCK(A);** | |
| | **X-LOCK(A);** |
| | write(A); |
| | **UNLOCK(A);** |
| **S-LOCK(A);** | |
| read(A); | |
| **UNLOCK(A);** | |

Figure 17: Schedule 6. Inconsistent state for transaction $T_1$

- If a transaction is waiting for a lock that is not granted, the transaction is *starved* and cannot progress. For example, schedule 7 shown in Fig. 18. $T_1$ has a $S-LOCK$ granted, while $T_2$ requests the lock but is waiting to get granted access to $A$. However, if starvation is not considered it will allow for $T_3$ to be granted the lock. Thus, even if $T_1$ has unlocked $A$, $T_2$ is still waiting. This can further happen with transaction $T_4, \cdots, T_n$. Thus, $T_2$ cannot progress as it is left waiting by the lock manager for the $X-LOCK$ on $A$. Starvation can be handled if a lock request is not blocked by a later request.

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $\cdots$ |
|---|---|---|---|---|
| **S-LOCK(A);** | | | | $\cdots$ |
| | **X-LOCK(A);** | | | $\cdots$ |
| | | **S-LOCK(A);** | | $\cdots$ |
| **UNLOCK(A);** | | | | $\cdots$ |
| | | | **S-LOCK(A);** | $\cdots$ |
| | | **UNLOCK(A);** | | $\cdots$ |

Figure 18: Schedule 7. Starvation of transaction $T_2$

- If two transactions are waiting for the other to release a lock, a *deadlock* can occur. Fig. 19 shows an example of a schedule that generates a deadlock. When $T_2$ ask for an $S-LOCK$ on $A$ it is not granted as $T_1$ still holds an $X-LOCK$ on $A$. The deadlock is then generated when $T_1$ asks for an $X-LOCK$ on $B$ as it cannot be granted due to $T_2$ having an $S-LOCK$ on $B$. Therefore, neither $T_1$ or $T_2$ can advance. Deadlocks are a necessary evil as they can be handled by rolling back transactions (Section 8).

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

Figure 19: Schedule 8. Deadlock of data items $A$ and $B$ between two transactions

*Locking protocols* define a set of rules that must be followed to lock and unlock data items. These protocols restrict the possible schedules and do not produce all possible serializable schedules. In the following subsections, these protocols will be defined. The relationship between these protocols and how long the locks are held is shown in Fig. 20.

| | $S-LOCK$ | $X-LOCK$ |
|---|---|---|
| **2PL** | short | short |
| **Strict 2PL** | short | long |
| **Rigorous 2PL** | long | long |

Figure 20: Lock liberation of 2PL protocols

## Two-phase locking

The *two-phase locking protocol* (2PL) or *basic 2PL* ensures serializability by issuing locks and unlocks in two phases:

1. **Growing or expanding phase.** Transactions start by obtaining locks without releasing them.
2. **Shrinking phase.** After locks are no longer needed, the transaction releases locks and cannot obtain any new ones.

Both $S-LOCK$s and $X-LOCK$s are *short locks*. A schedule example is shown in Fig. 21. First, both transactions $T_1$ and $T_2$ ask for all the locks for all the data items used at the beginning of the transaction. For $T_1$, there is an $X-LOCK$ for $A$ and $B$, while for $T_2$ it is a $X-LOCK$ for $A$. After $T_1$ completely finishes using the $X-LOCK$ on $A$, it is released thus $T_2$ can continue executing the operations. Furthermore, as $A$ is not used any more in $T_1$, there will be no inconsistent data states.

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| **X-LOCK(B);** | |
| read(A); | |
| write(A); | |
| | **X-LOCK(A);** |
| **UNLOCK(A);** | |
| | read(A); |
| | write(A); |
| read(B); | |
| write(B); | |
| **ABORT;** | |

Figure 21: Schedule 9. 2PL with cascading aborts

However, there are several limitations to 2PL. Deadlocks can still occur and it limits concurrency. Furthermore, *cascading aborts* or *cascading rollbacks* can happen when a transaction aborts and another transaction must be rolled back. For example, in schedule 9 of Fig. 21 there is an abort at the end of $T_1$, thus $T_2$ must also abort the changes. Thus the effort of executing the instructions in $T_2$ is wasted.

## Strict two-phase locking

To avoid *cascading rollbacks*, we can use a modification of 2PL called *strict two-phase locking protocol*. The $X-LOCK$s are only released until the transaction commits or aborts. Therefore, $S-LOCK$s are *short locks* and $X-LOCK$s are *long locks*. For example, Fig. 22 shows schedule 10 where all the locks are requested at the beginning of the execution of $T_1$ and $T_2$. $T_1$ can release the $S-LOCK$ on $B$ before the end of the transaction, however the $X-LOCK$ on $A$ is released when the transaction will be committed. Until then can the $X-LOCK$ on $A$ be granted for $T_2$ eliminating the cascading rollbacks.

## Rigorous two-phase locking

Another variant is called *rigorous or strong strict two-phase locking protocol* where all the locks are held until the transaction commits. Thus, $S-LOCK$s and $X-LOCK$s are *long locks*. Strict or rigorous 2PL eliminates cascading aborts, but the schedules limit concurrency.

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| **S-LOCK(B);** | |
| | **X-LOCK(A);** |
| read(B); | |
| **UNLOCK(B);** | |
| read(A); | |
| write(A); | |
| **UNLOCK(A);** | |
| **COMMIT;** | read(A); |
| | write(A); |
| | **COMMIT;** |

Figure 22: Schedule 10. Strict 2PL without cascading aborts

# 8 Deadlock handling

A deadlock occurs when there is a set of transactions where each transaction is waiting for a lock that another transaction in the set holds. No transaction in the set can progress. We can handle deadlocks by detection and recovery, or prevention.

## Deadlock detection and recovery

*Deadlock detection* and *recovery* protocols allow the system to enter a deadlock state that will be recovered. To determine if a deadlock occured, periodically a deadlock detection algorithm is invoked and when detected the system uses recovery algorithms. If the probability of deadlock is not high, these techniques are more efficient than deadlock prevention. But, these techniques have an additional overhead of maintaining information and executing the algorithms.

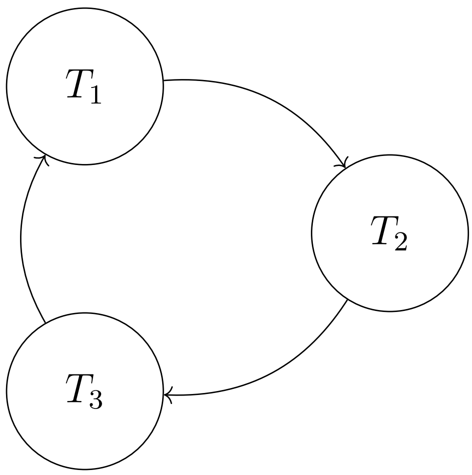
Deadlocks can be detected with *waits-for graphs*. Every transaction is a node. An edge is created from $T_i$ to $T_j$ if $T_i$ is waiting for transaction $T_j$ to release a lock. The edge is removed when $T_j$ releases the data item required by $T_i$. A deadlock exists in the system if, when we check periodically, there is a cycle in the wait-for graph. The deadlock detection algorithm is invoked frequently if many deadlocks occur. Worst case, the algorithm could be invoked every time a lock is not granted immediately.

An example of a wait-for graph is shown in Fig. 23. The subfigure 23a shows the schedule, while the subfigure 23b.

- The edge $T_1 \rightarrow T_2$ is created as $T_1$ waits for the $S - LOCK$ on $B$ that $T_2$ holds.
- The edge $T_3 \rightarrow T_2$ is **not** created as both $T_2$ and $T_3$ can hold the $S - LOCK$ on $D$.
- The edge $T_2 \rightarrow T_3$ is created as $T_2$ waits for the $S - LOCK$ on $c$ that $T_3$ holds.
- The edge $T_3 \rightarrow T_1$ is created as $T_1$ waits for the $X - LOCK$ on $A$ that $T_1$ holds.

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| **S-LOCK(A);** | | |
| | **X-LOCK(B);** | |
| | | **X-LOCK(C);** |
| | **S-LOCK(D);** | |
| | | **S-LOCK(D);** |
| **S-LOCK(B);** | | |
| | **S-LOCK(C);** | |
| | | **X-LOCK(A);** |
| ... | ... | ... |

(a) Schedule with three-way deadlock




(b) Wait-for graph to the schedule

Figure 23: Deadlock represented in a wait-for graph

When a deadlock is detected, the system must *recover*. To do this, we must determine which transactions must be rolled back to break the deadlock.

The set of transactions to roll back must be selected (*victim selection*) in such a way that will incur the minimum cost. There are several factors to consider such as the age, progress, number of data items used and number of transactions involved in the rollback. A combination of several factors are considered. We must include the number of rollbacks as a cost factor to not select only one victim.

When we decide the victim transaction, it has to determine how far the transaction needs to be rolled back. A *total rollback* aborts all the transaction and restarts it. A *partial rollback*, using the sequence of grants and requests of locks with the updates, can determine a point to revert the changes to resume the execution from there.

## Deadlock prevention

*Deadlock prevention* protocols ensure that the system will never enter a deadlock state. These schemes are used if the probability of deadlock is high. Several approaches have been proposed.

### Avoiding cyclical waits approaches

The first approach focuses on not allowing cyclical waits. The simplest way to achieve this is to acquire all the locks at the beginning of the transaction execution in one step, but the data-item utilization is low and is difficult to predict before initialization. Furthermore, when the transaction cannot acquire all the locks it must wait. An example can be seen in Fig. 24 where $T_2$ must wait as it could not acquire all the locks, while $T_1$ has all the locks. $T_2$ can only continue when all the locks required are released.

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A,B);** | |
| ... | **X-LOCK(A,B);** |
| **UNLOCK(A);** | |
| ... | |
| **UNLOCK(B);** | |
| COMMIT; | ... |

Figure 24: Deadlock prevention acquiring all locks in one step

Another one of these approaches consists of ordering all the data items and strictly following the ordering. A variation considering both of these schemes is to use 2PL with strict ordering, thus the locks can only be requested following the order. An example can be seen in Fig. 25. The locks are defined to be acquired with $A$ first and then $B$. A deadlock does not occur when $T_2$ tries to acquire $A$, the transaction must wait. If we had no order, $T_2$ could acquire the $X - LOCK(B)$ first and a deadlock would happen.

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | **X-LOCK(A);** |
| **X-LOCK(B);** | |
| ... | |
| **UNLOCK(B);** | |
| **UNLOCK(A);** | |
| COMMIT; | **X-LOCK(B);** |
| | ... |

Figure 25: Deadlock prevention 2PL with strict ordering

### Roll back approaches

Furthermore, there is a roll back approach. When a transaction tries to acquire a lock held by another transaction (possible deadlock), one of the transactions will be rolled back. To decide which transaction to rollback a unique timestamp is assigned to each transaction when it begins, prioritizing older transactions. The rolled back transactions retain it's initial timestamp. There are two different timestamp-based schemes:

- **Wait-die ("Old waits for young"):** If the requesting transaction has a higher priority than the holding transaction, it waits. Otherwise, it is aborted. The older transaction is allowed to *wait* for a younger transaction. The younger transaction *dies* (aborts) if it requests the lock held by an older transaction.
- **Wound-wait ("Young waits for old"):** If the requesting transaction has a higher priority than the holding transaction, the holding transaction aborts and rolls back (wounds). Otherwise, it waits. The younger transaction is allowed to *wait* for an older transaction. The older transaction *wounds* (abort) the younger transaction holding the lock.

An example for both timestamped-based schemes is shown in Fig. 26. For example in Subfigure. 26a $T_1$ starts before $T_2$, thus $T_1 < T_2$ ($T_1$ is older). When $T_1$ requests a lock held by $T_2$, for wait-die the older transaction $T_1$ waits for transaction $T_2$ to release the lock. If the protocol was wound-wait then $T_2$ is aborted and rolled back. For another example shown in Subfigure. 26a $T_1$ also starts before $T_2$ ($T_1$ is older). With the wait-die scheme when $T_2$ requests the lock held by $T_1$, $T_2$ dies. While, for the wound-die scheme $T_2$ can wait for the older transaction.

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| | BEGIN |
| | **X-LOCK(A);** |
| **X-LOCK(A);** | |

(a) Wait-die $T_1$ waits and wound-wait $T_2$ aborts

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| **X-LOCK(A);** | |
| | BEGIN |
| | **X-LOCK(A);** |

(b) Wait-die $T_2$ waits and wound-wait $T_1$ aborts

Figure 26: Deadlock prevention based on timestamps

Both timestamped-based techniques prevent deadlocks as either transactions wait for the younger (wait-die) or older transactions (wound-wait), therefore no cycle is created. However, both techniques have many unnecessary rollbacks.

Another method is using *lock timeouts*, defining a specified amount for waiting. If the transaction was not granted the lock after waiting the specified time, it will roll back and restart. Defining the time to wait is difficult, long times cause unnecessary delays while short waits lead to wasted results. This protocol falls between deadlock prevention and detection. Starvation can occur, thus there is limited applicability to the scheme.

# 9 Multiple granularity

Locking many single data items is costly, as it requires the lock manager to request many locks. However, instead of only locking a single data item, we could consider different types of *granularity.* This way *smaller* or *finer* granularities, such as attributes, can be requested by locking a *larger* or *coarser* granularity item, such as a table. Large granularities allow for less concurrency, while smaller granularities have a higher overhead with the lock manager. The level of granularity used will depend on the type of transactions involved and will be selected by the DBMS.

The size and hierarchy of the different data granularity can be defined as a tree. An example of such a tree is shown in Fig. 27. Every node is an independent data item. The highest level is all the databases. The second level is the tables saved in the database. The third level are tuples of the respective table. Lastly, the fourth level represents the attributes of the respective tuple. If we acquire a lock on Table 1 (*explicit lock*), then all the descendants such as Tuple 1, Tuple 2, $\cdots$, Tuple, Attribute 1, Attribute 2, $\cdots$ and Attribute p will acquire the same lock (*implicit lock*). The granularity hierarchy tree could also have the following four levels: database, areas, files and records.

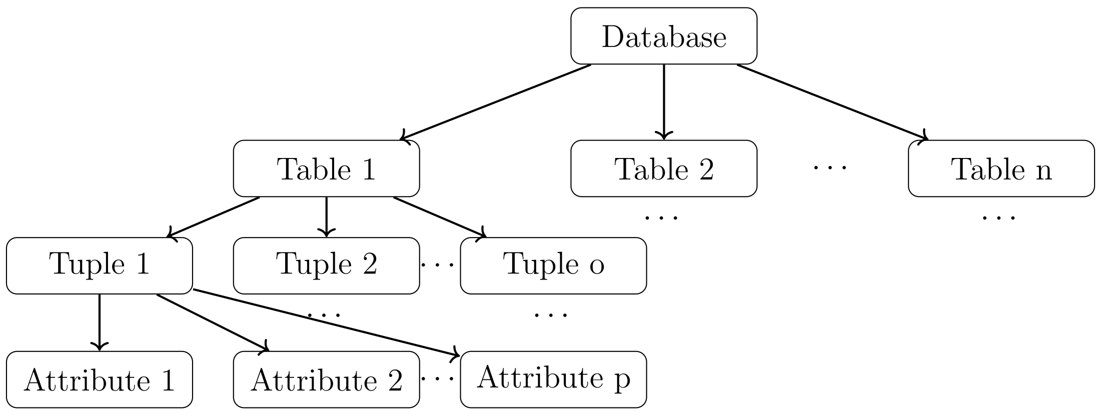



Figure 27: Granularity hierarchy

To determine if a lock can be granted we have to check if all the implicit locks are compatible with requested mode. To determine this, all the descendants of the granularity level must be checked. However, this is not efficient. In the worst case, locking the database requires checking all the nodes in the tree.

To check more efficiently, we can use *intention lock modes* to not have to check all descendant nodes. With intention mode, all the descendants are explicitly locked. The different modes allowed are the following:

- **Intention-shared (IS) mode:** Indicares that the descendants have explicit shared-mode locks.
- **Intention-exclusive (IX) mode:** Indicates that the descendants have explicit exclusive-mode or shared-mode locks.
- **Shared and intention-exclusive (SIX) mode:** The node is explicitly locked in shared-mode, with the descendants explicitly locked in exclusive-mode.

The compilability matrix of the modes are shown in Fig. 28.

The *multiple-granularity locking protocol* can ensure serializability by setting the lock at the highest level of the database hierarchy. For example, assume a hierarchy tree for a database that saves Students' records. We will only exemplify the protocol using the second (table) and third level (tuple) of the hierarchy tree. The final results of the locks are shown in Fig. 29.

| | $IS$ | $IX$ | $S$ | $SIX$ | $X$ |
|---|---|---|---|---|---|
| $IS$ | ✓ | ✓ | ✓ | ✓ | ✗ |
| $IX$ | ✓ | ✓ | ✗ | ✗ | ✗ |
| $S$ | ✓ | ✗ | ✓ | ✗ | ✗ |
| $SIX$ | ✓ | ✗ | ✗ | ✗ | ✗ |
| $X$ | ✗ | ✗ | ✗ | ✗ | ✗ |

Figure 28: Compatibility matrix of intention lock modes

- Suppose that transaction $T_1$ wants to read the data for Alice. Therefore, an $S - LOCK$ is acquired for the tuple with Alice's data and a $IS - LOCK$ on the Student table indicating that one of its descendants (Alice's tuple) has an $S - LOCK$.
- Now, let us assume that transaction $T_2$ wants to update Carlos's record. We would require to acquire an $X - LOCK$ for Carlos' tuple and an $IX - LOCK$ on the Student table. We can acquire the lock on the table, as the $IS - LOCK$ is compatible with the $IX - LOCK$.
- Finally, let's assume that transaction $T_3$ scans wants to scan the student table to update some student records. This would require gaining a $SIX - LOCK$ on the student table. However, this would not be possible as $SIX$ is not compatible with $IX$. Therefore, $T_3$ would have to wait to be granted the lock. If $T_3$ happend before $T_2$, we could have acquired the lock for $T_3$.

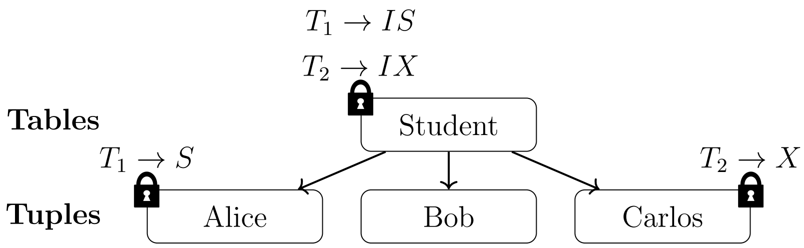



Figure 29: Result of applying the multiple granularity protocol

Locks must be acquired from the root to the leaf, and released from the left to the root. This protocol enhances concurrency and reduces overhead, but deadlocks can still occur.

# 10 Timestamp-based protocols

The lock based protocols define the order at execution time, but we could also define the serializability order in advance using a *timestamp-ordering scheme.* A unique timestamp $TS(T_i)$ for every transaction $T_i$ before execution.

When a new transaction $T_j$ enters the system then $TS(T_i) < TS(T_j)$. To assign the sequential timestamp we could use different strategies. The *system clock* can be used to assign a value when the transaction enters the system, however there can be issues in edge cases (e.g., daylight savings). Another option is to use a *logical counter* that increments after a new transaction enters the system, but the counter could overflow or has issues maintaining the counter across multiple machines. A hybrid combination of methods can be used. If $TS(T_i) < TS(T_j)$ the DBMS must ensure an equivalent serial execution of $T_i$ appearing before $T_j$.

The *timestamp-ordering protocol* (basic T/O) executes conflicting *reads* and *write* operations in the timestamp order, ensuring conflict serializability. To execute the scheme, the DBMS tracks for every data item $Q$ the last transaction that executed successfully a *read* ($R - TS(Q)$) and a *write* ($W - TS(Q)$). The DBMS updates these timestamps after every instruction is executed.

- **Read operations.**
  - If $TS(T_i) < W - TS(Q)$, a future transaction has written to data item $Q$ before $T_i$ violating the $TS(T_i) < TS(T_j)$ property. Therefore, $T_i$ is aborted and restarted with a new timestamp value.
  - Else, $TS(T_i) \geq W - TS(Q)$ the order $TS(T_i) < TS(T_j)$ is preserved and the $read(Q)$ instruction is executed. The DBMS also updates $R-TS(Q) = max(R-TS(Q), TS(T_i))$.
- **Write operations.**
  - If $TS(T_i) < R-TS(Q)$ or $TS(T_i) < W-TS(Q)$, a future transaction has read or written to data item $Q$ before $T_i$ violating the $TS(T_i) < TS(T_j)$ property. Therefore, $T_i$ is aborted and restarted with a new timestamp value.
  - Else, $TS(T_i) \geq R - TS(Q)$ and $TS(T_i) \geq W - TS(Q)$ the order of execution is ensured and the $write(Q)$ instruction is executed. The DBMS also updates $W-TS(Q) = TS(T_i)$.

The timestamped protocol example will use the schedule in Fig. 30. We can assume that $TS(T_1) = 1$ and $TS(T_2) = 2$.

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

Figure 30: Schedule for basic T/O example

- When $T_1$ executes $read(B)$, $W-TS(B) = 0$. As$TS(T_1) \geq W-TS(B) = 1 \geq 0$, $T_1$ can execute the instruction and $R - TS(B) = max(R - TS(B), TS(T_1)) = max(0, 1) = 1$.
- When $T_2$ executes $read(B)$, $W - TS(B) = 0$. As $TS(T_2) \geq W - TS(B) = 2 \geq 0$, $T_2$ can execute the instruction and $R - TS(B) = max(R - TS(B), TS(T_2)) = max(1, 2) = 2$.

- When $T_2$ executes $write(B)$, $W - TS(B) = 0$ and $R - TS(B) = 2$. As $TS(T_2) \geq W - TS(B) = 2 \geq 0$ and $TS(T_2) \geq R - TS(B) = 2 \geq 2$, $T_2$ can execute the instruction and $W - TS(B) = TS(T_2) = 2$.
- When $T_1$ executes $read(A)$, $W - TS(A) = 0$. As $TS(T_1) \geq W - TS(A) = 1 \geq 0$, $T_1$ can execute the instruction and $R - TS(A) = max(R - TS(A), TS(T_1)) = max(0, 1) = 1$.
- When $T_2$ executes $read(A)$, $W - TS(A) = 0$. As $TS(T_2) \geq W - TS(A) = 2 \geq 0$, $T_2$ can execute the instruction and $R - TS(A) = max(R - TS(A), TS(T_2)) = max(1, 2) = 2$.
- When $T_1$ executes $read(A)$, $W - TS(A) = 0$. As $TS(T_1) \geq W - TS(A) = 1 \geq 0$, $T_1$ can execute the instruction and $R - TS(A) = max(R - TS(A), TS(T_1)) = max(2, 1) = 2$.
- When $T_2$ executes $write(A)$, $W - TS(A) = 0$ and $R - TS(A) = 2$. As $TS(T_2) \geq W - TS(A) = 2 \geq 0$ and $TS(T_2) \geq R - TS(a) = 2 \geq 2$, $T_2$ can execute the instruction and $W - TS(A) = TS(T_2) = 2$.

The protocol can also be modified with *Thomas' write rule* to remove unnecessary rollbacks for write, creating view serializable conflicts. If $TS(T_i) < W - TS(Q)$ the system can ignore the write as it is obsolete. This does violate the timestamp order, but it is fine as no other transaction will read the $write(Q)$ of $T_i$.

The protocol ensures conflict serializability and freedom from deadlocks, though it has issues due to starvation, non-recoverable schedules and phantom tuples. Some of these problems may be fixed with variations of the protocol.

# 11 Validation-based protocols

When most transactions only invoke *read* the conflicts are low, thus the concurrency control schemes may provide unnecessary overhead. When a transaction reads a value integrity is never lost, thus only writing will we require *validating* the consistency of the database. There are three phases of execution for writes, shown in Fig. 31, that must be executed sequentially for a transaction to ensure the consistency.

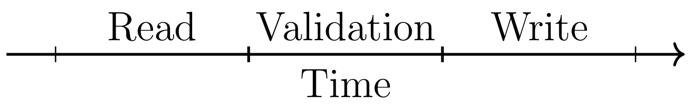
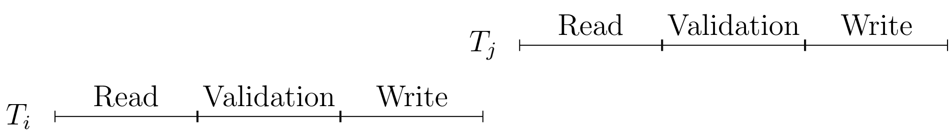



Figure 31: Validation protocol transaction execution phases (only writes)

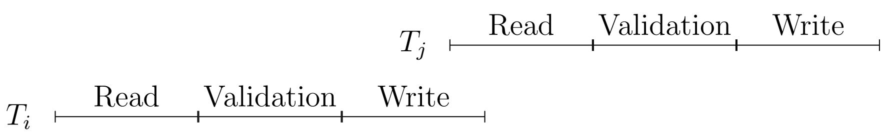
1. **Read phase.** The data is *read* to a temporary copy. The transaction the modified the temporary results (not global).
2. **Validation phase.** The system *validates* whether a serializability conflict occurred. If there is a violation the transaction is aborted.
3. **Write phase.** The temporary results are *writen* to the database (global).

To validate the serial equivalence, each transaction $T_i$ will be assigned an order of execution based on a timestamp $TS(T_i)$. If $T_i$ is older than $T_j$ then $TS(T_i) < TS(T_j)$. To ensure the equivalence the serial equivalence for any younger transaction $T_j$ for every older transaction $T_i$ one of following conditions must hold (Fig. 32):

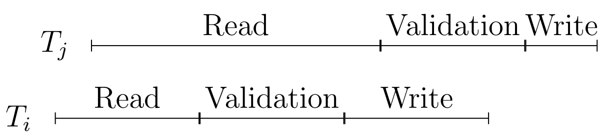
1. $T_i$ completes all phases before $T_j$ begins (Subfig. 32a).
2. $T_i$ completes all phases before $T_j$ enters the validation phase and the transactions do not read the same items (Subfig. 32b).
3. $T_i$ completes the read phase before $T_j$ and the transactions do not read or write the same items (Subfig. 32c).




(a) $T_i$ finishes before $T_j$ begins




(b) $T_i$ finishes before $T_j$ enters validation




(c) $T_i$ completes the read phase before $T_j$

Figure 32: Validation order of transactions for validation protocol

This approach is optimistic as the validation will fail only in the worst of cases. The protocol automatically guards against rollbacks and has no deadlocks, however starvation can happen.

# 12 Isolation levels

Programmers can decide the appropriate level of concurrency required. *Isolation levels* control the extent of isolation between transactions. Depending on the application, a weaker isolation level may improve the system performance but increases the risk of inconsistency.

Different DBMS implement different isolation levels, however most of them are based from the *SQL-92 standard* that defines the following isolation levels based on transaction problems: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The degree of transaction concurrency versus database consistency for each isolation level is shown in Fig. 33.

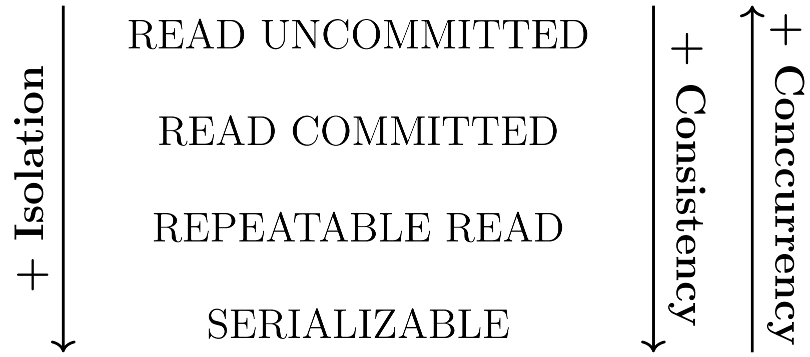



Figure 33: Transaction concurrency versus database consistency with SQL-92 isolation levels

The transaction problems allowed by isolation level are shown in Fig. 34. An x-mark (✗) indicates that the problem cannot occur, while a question mark (?) indicates that it might happen.

| | Lost update | Dirty read | Unrepeatable read | Phantom read |
|---|---|---|---|---|
| READ UNCOMMITTED | ✗ | ? | ? | ? |
| READ COMMITTED | ✗ | ✗ | ? | ? |
| REPEATABLE READ | ✗ | ✗ | ✗ | ? |
| SERIALIZABLE | ✗ | ✗ | ✗ | ✗ |

Figure 34: Isolation levels with transaction problems allowed

The SQL-92 isolation levels can be implemented using 2PL-lock based systems. The length of holding each lock is shown in Fig. 35. All isolation levels hold long $X-LOCK$s to ensure that no *lost update* issues occur. While depending on the isolation level the length and type of lock held for $S-LOCK$s vary. Locks for data-items holds the particular data-item, while locks for a condition lock all elements that satisfy the condition. Not all DBMS isolation levels that are not analogous to locking protocols, thus new systems may consider other concurrency management protocols.

| | $S-LOCK$ | | $X-LOCK$ |
|---|---|---|---|
| | data-item | condition | |
| READ UNCOMMITTED | None | None | Long |
| READ COMMITTED | Short | Short | Long |
| REPEATABLE READ | Long | Short | Long |
| SERIALIZABLE | Long | Long | Long |

Figure 35: Lock liberation for isolation levels

Most DBMS run the **READ COMMITTED** isolation level by default. We can change the isolation level in *SQL SERVER* with the **SET TRANSACTION ISOLATION LEVEL <NAME>**. Furthermore, by default most DBMS commit every statement after execution (*automatic commit*). To enable manual commits, the transaction begins with the start transaction command (in SQL Server it starts with **BEGIN TRANSACTION**) and ends with a **COMMIT** or **ROLLBACK**.

# References

[1] R. Elmasri and S. Navathe, *Fundamentals of database systems*, 7th ed. Pearson, 2016, chapters 20 and 21.

[2] A. Crotty and M. Li. Lectures #15 to #18. [Online]. Available: https://15445.courses.cs.cmu.edu/fall2021/schedule.html

[3] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York, NY: McGraw-Hill, 2020, chapters 17 and 18.

[4] H. Berenson, "A Critique of ANSI SQL Isolation Levels," p. 10.

[5] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, vol. 6, no. 2, p. 14.