CS W186
Spring 2020

Transactions

1 Introduction

In reality, we usually don't have just one person accessing a database at a time. Many users can make requests to a database at a time which can cause concurrency issues. What happens when one user writes and then another user reads from the same resource? What if both users try to write to the same resource? There are several problems we can run into when several users are using the database at the same time if we're not careful:

- **Inconsistent Reads**: A user reads only part of what was updated.
  - User 1 updates Table 1 and then updates Table 2.
  - User 2 reads Table 2 (which User 1 has not updated yet) and then Table 1 (which User 1 already updated) so it reads the database in an intermediate state.
- **Lost Update**: Two users try to update the same record so one of the updates gets lost. For example:
  - User 1 updates a toy's price to be price * 2.
  - User 2 updates a toy's price to be price + 5, blowing away User 1's update.
- **Dirty Reads**: One user reads an update that was never committed.
  - User 1 updates a toy's price but this gets aborted.
  - User 2 reads the update before it was rolled back.

2 Transactions

Our solution to those problems is to define a set of rules and guarantees about operations. We will do this by using **transactions**. A transaction[1] is a sequence of multiple actions that should be executed as a single, logical, atomic unit. Transactions guarantee the **ACID properties** to avoid those problems:

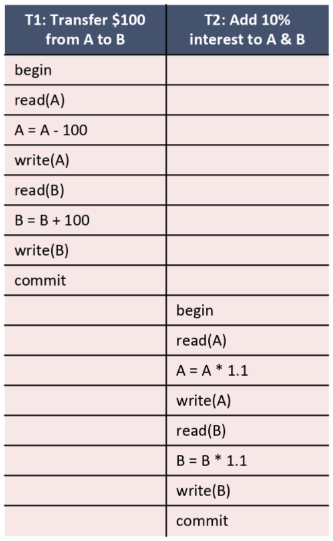
- **Atomicity**: A transaction ends in two ways: it either **commits** or **aborts**. Atomicity means that either all actions in the Xact happen, or none happen.
- **Consistency**: If the DB starts out consistent, it ends up consistent at the end of the Xact.
- **Isolation**: Execution of each Xact is isolated from that of others. In reality, the DBMS will interleave actions of many Xacts and not execute each in order of one after the other. The DBMS will ensure that each Xact executes as if it ran by itself.
- **Durabilty**: If a Xact commits, its effects persist. The effects of a committed Xact must survive failures.

3 Concurrency Control

In this note we will discuss how to enforce the **isolation** property of transactions (we will learn how the other properties are enforced in the note about recovery). To do this, we will analyze **transaction schedules** which show the order that operations are executed in. These operations include: Begin, Read, Write, Commit and Abort.

The easiest way to ensure isolation is to run all the operations of one transaction to completion before beginning the next transaction. This is called a **serial schedule**. For example, the following schedule is a serial schedule because $T1$'s operations run completely before $T2$ runs.

[1] We sometimes shorten transaction to Xact.

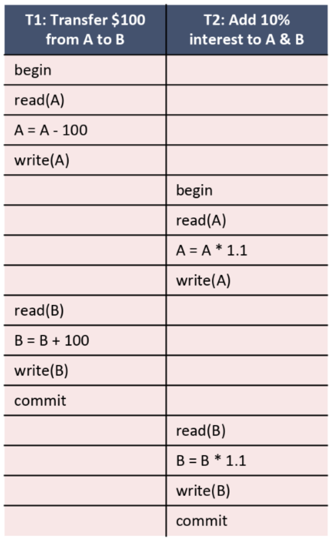


T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit

The problem with these schedules, however, is that it is not efficient to wait for an entire transaction to finish before starting another one. We want to get the same results as a serial schedule (because we know serial schedules are correct) while getting the performance benefits of running schedules simultaneously. Basically, we are looking for a schedule that is **equivalent** to a serial schedule. For schedules to be equivalent they must satisfy the following three rules:

1. They involve the same transactions
2. Operations are ordered the same way within the individual transactions
3. They each leave the database in the same state

If we find a schedule whose results are equivalent to a serial schedule, we call the schedule **serializable**. For example, the following schedule is serializable because it is equivalent to the schedule above. You can work through the following schedule and see that resources A and B end up with the same value as the serial schedule above.



T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
	begin
	read(A)
	A = A * 1.1
	write(A)
read(B)	
B = B + 100	
write(B)	
commit	
	read(B)
	B = B * 1.1
	write(B)
	commit

Now the question is: how do we ensure that two schedules leave the database in the same final state without running through the entire schedule to see what the result is? We can do this by looking for **conflicting operations**. For two operations to conflict they must satisfy the following three rules:

1. the operations are from different transactions
2. both operations operate on the same resource
3. at least one operation is a write

We then check if the two schedules order every pair of conflicting operations in the same way. If they do, we know for sure that the database will end up in the same final state. When two schedules order their conflicting operations in the same way the schedules are said to be **conflict equivalent**, a stronger condition than being equivalent.

Now that we have a way of ensuring that two schedules leave the database in the same final state, we can check if a schedule is conflict equivalent to a serial schedule without running through the entire schedule. We call a schedule that is conflict equivalent to some serial schedule **conflict serializable**. Note: if a schedule S is conflict serializable then it implies that is serializable.[2].

3.1 Conflict Dependency Graph

Now we have a way of checking if a schedule is serializable! We can check if the schedule is conflict equivalent to some serial schedule because conflict serializable implies serializable. We can check

[2] Not all serializable schedules are conflict serializable

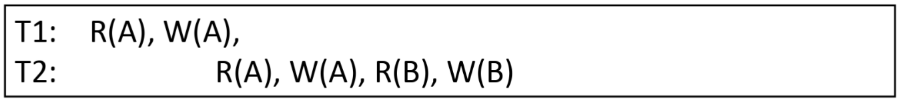
conflict serializability by building a **dependency graph**. Dependency graphs have the following structure:

- One node per Xact
- Edge from Ti to Tj if:
    - an operation Oi of Ti conflicts with an operation Oj of Tj
    - Oi appears earlier in the schedule than Oj

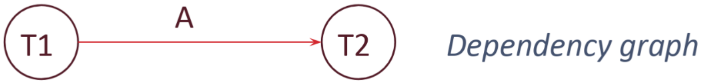
A schedule is conflict serializable if and only if its dependency graph is acyclic. So all we have to do is check if the graph is acyclic to know for sure that it is serializable!

Let's take a look at two examples:

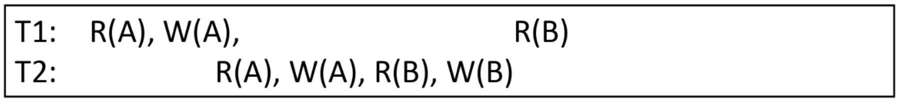
- The following schedule is conflict serializable and the conflict graph is acyclic. There are two conflicting operations:
  - $T1$ reads A and then $T2$ writes to A. Because of this, there will be an edge from $T1$ to $T2$.
  - $T1$ writes to A and then $T2$ reads from A. Since there already is an edge from $T1$ to $T2$, we don't have to add the edge again.



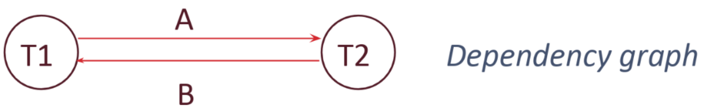
```
T1:   R(A), W(A),
T2:               R(A), W(A), R(B), W(B)
```



- The following schedule is not conflict serializable and the conflict graph is not acyclic. Some conflicting operations:
  - $T1$ reads A and then $T2$ writes to A. Because of this, there will be an edge from $T1$ to $T2$.
  - $T2$ writes to B and then $T1$ reads B. Because of this, there will be an edge from $T2$ to $T1$.



```
T1:   R(A), W(A),                           R(B)
T2:               R(A), W(A), R(B), W(B)
```



CS W186
Spring 2020

Transactions

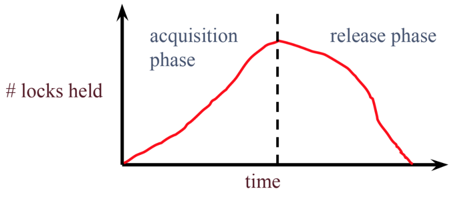
4 Two Phase Locking

What are locks, and why are they useful? Locks are basically what allows a transaction to read and write data. For example, if Transaction $T1$ is reading data from resource A, then it needs to make sure no other transaction is modifying resource A at the same time. So a transaction that wants to read data will ask for a Shared (S) lock on the appropriate resource, and a transaction that wants to write data will ask for an Exclusive (X) lock on the appropriate resource. Only one transaction may hold an exclusive lock on a resource, but many transactions can hold a shared lock on data. **Two phase locking (2PL)** is a scheme that ensures the database uses conflict serializable schedules. The two rules for 2PL are:

- Transactions must obtain a S (shared) lock before reading, and an X (exclusive) lock before writing.
- Transactions cannot get new locks after releasing any locks – this is the key to enforcing serializability through locking!

Lock Compatibility Matrix

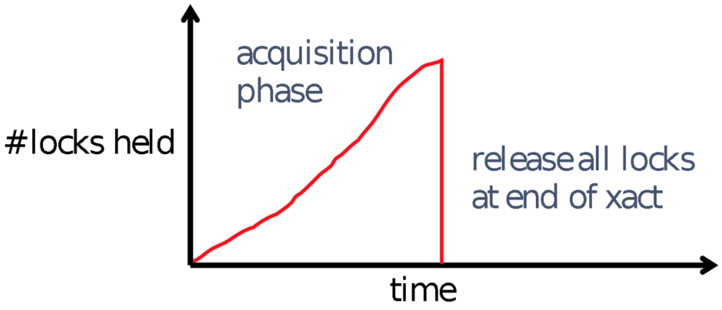
	S	X
S	√	–
X	–	–



The problem with this is that it does not prevent **cascading aborts**. For example,

- $T1$ updates resource A and then releases the lock on A.
- $T2$ reads from A.
- $T1$ aborts.
- In this case, $T2$ must also abort because it read an uncommitted value of A.

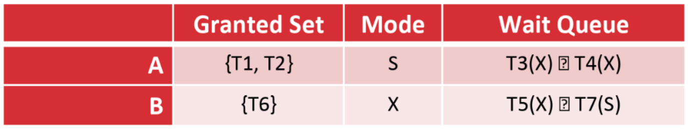
To solve this, we will use **Strict Two Phase Locking**. Strict 2PL is the same as 2PL, except all locks get released together when the transaction completes.



5 Lock Management

Now we know what locks are used for and the types of locks. We will take a look at how the Lock Manager[3] manages these lock and unlock requests and how it decides when to grant the lock.

The LM maintains a hash table, keyed on names of the resources being locked. Each entry contains a granted set (a set of granted locks/the transactions holding the locks for each resource), lock type (S or X or types we haven't yet introduced), and a wait queue (queue of lock requests that cannot yet be satisfied because they conflict with the locks that have already been granted). See the following graphic:



	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) □ T4(X)
B	{T6}	X	T5(X) □ T7(S)

When a lock request arrives, the Lock Manager checks if any Xact in the Granted Set or in the Wait Queue want a conflicting lock. If so, the requester gets put into the Wait Queue. If not, then the requester is granted the lock and put into the Granted Set.

In addition, Xacts can request a lock upgrade: this is when a Xact with shared lock can request to upgrade to exclusive. The Lock Manager will add this upgrade request at the front of the queue.

[3] We will refer to the Lock Manager as LM sometimes.

Here is some pseudocode for how to process the queue; note that it doesn't explicitly go over what to do in cases of promotion etc, but it's a good overview nevertheless.

```
# If queue skipping is not allowed, here is how to process the queue

H = set of held locks on A
Q = queue of lock requests for A

def request(lock_request):
    if Q is empty and lock_request is compatible with all locks in H:
        grant(lock_request)
    else:
        addToQueue(lock_request)

def release_procedure(lock_to_release):
    release(lock_to_release)
    for lock_request in Q:        # iterate through the lock requests in order
        if lock_request is compatible with all locks in H:
            grant(lock_request)   # grant the lock, updating the held set
        else:
            return
```

Note that this implementation does not allow **queue skipping**. When a request arrives under a queue skipping implementation, we first check if you can grant the lock based on what locks are held on the resource; if the lock cannot be granted, then put it at the back of the queue. When a lock is released and the queue is processed, grant *any* locks that are compatible with what is currently held.

For an example of queue skipping and pseudocode, see the appendix. It relies on you understanding multigranulariy locking however, so make sure to read section 7 first to understand the example.

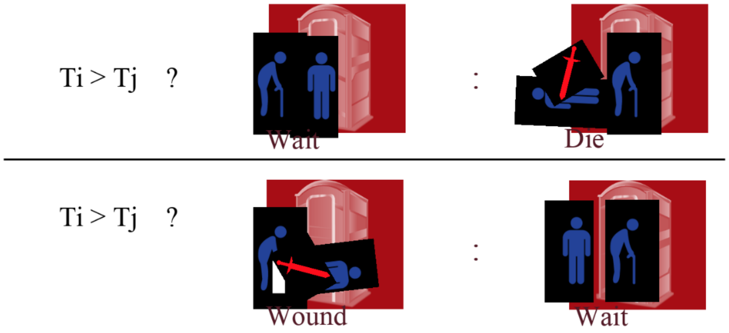
6 Deadlock

We now have a lock manager that will put requesters into the Wait Queue if there are conflicting locks. But what happens if $T1$ and $T2$ both hold S locks on a resource and they both try upgrade to X? $T1$ will wait for $T2$ to release the S lock so that it can get an X lock while $T2$ will wait for $T1$ to release the S it can get an X lock. At this point, neither transaction will be able to get the X lock because they're waiting on each other! This is called a **deadlock**, a cycle of Xacts waiting for locks to be released by each other.

6.1 Avoidance

One way we can get around deadlocks is by trying to **avoid** getting into a deadlock. We will assign the Xact's **priority** by its age: now - start time. If Ti wants a lock that Tj holds, we have two options:[4]

- **Wait-Die**: If Ti has higher priority, Ti waits for Tj; else Ti aborts
- **Wound-Wait**: If Ti has higher priority, Tj aborts; else Ti waits



[4] Important Detail: If a transaction re-starts, make sure it gets its original timestamp.

6.2 Detection

Although we avoid deadlocks in the method above, we end up aborting many transactions! We can instead try detecting deadlocks and then if we find a deadlock, we abort one of the transactions in the deadlock so the other transactions can continue.

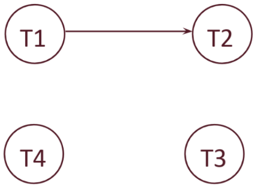
We will detect deadlocks by creating and maintaining a **"waits-for" graph**. This graph will have one node per Xact and an edge from T_i to T_j if:

- T_j holds a lock on resource X
- T_i tries to acquire a lock on resource X, but T_j must release its lock on resource X before T_i can acquire its desired lock.

For example, the following graph has a edge from $T1$ to $T2$ because after $T2$ acquires a lock on B, $T1$ tries to acquire a conflicting lock on it. Thus, $T1$ waits for $T2$.

Example:

```
T1:  S(A)  S(D)          S(B)
T2:      X(B)
T3:
T4:
```



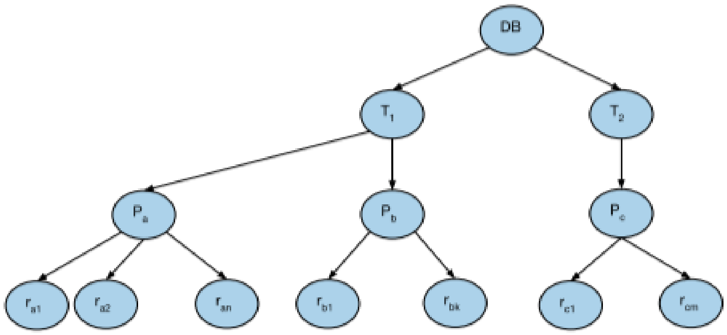
We will periodically check for cycles in a graph which indicates a deadlock. If a cycle is found - we will "shoot" a Xact in the cycle.

Important note: A "waits-for" graph is used for cycle detection and is different from the conflict dependency graph we discussed earlier which was used to figure out if a transaction schedule was serializable.

7 Lock Granularity

So now that we understand the concept of locking, we want to figure out what to actually lock. Do we want to lock the tuple containing the data we wish to write? Or the page? Or the table? Or maybe even the entire database, so that no transaction can write to this database while we're working on it? As you can guess, the decision we make will differ greatly based upon the situation we find ourselves in.

Let us think of the database system as the tree below:



The top level is the database. The next level is the table, which is followed by the pages of the table. Finally, the records of the table themselves are the lowest level in the tree.

Remember that when we place a lock on a node, we implicitly lock all of its children as well (intuitively, think of it like this: if you place a lock on a page, then you're implicitly placing a lock on all the records and preventing anyone else from modifying it). So you can see how we'd like to be able to specify to the database system exactly which level we'd really like to place the lock on. That's why multigranularity locking is important; it allows us to place locks at different levels of the tree.

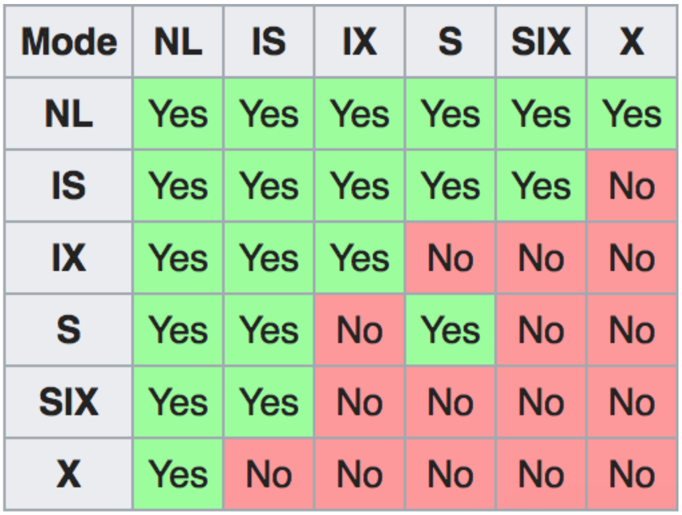
We will have the following new lock modes:

- IS: Intent to get S lock(s) at finer granularity.
- IX: Intent to get X lock(s) at finer granularity. Note: that two transactions can place an IX lock on the same resource – they do not directly conflict at that point because they could place the X lock on two different children! So we leave it up to the database manager to ensure that they don't place X locks on the same node later on while allowing two IX locks on the same resource.

- SIX: Like S and IX at the same time. This is useful if we want to prevent any other transaction from modifying a lower resource but want to allow them to read a lower level. Here, we say that at this level, I claim a shared lock; now, no other transaction can claim an exclusive lock on anything in this sub-tree (however, it can possibly claim a shared lock on something that is not being modified by this transaction–i.e something we won't place the X lock on. That's left for the database system to handle).

Interestingly, note that no other transaction can claim an S lock on the node that has a SIX lock, because that would place a shared lock on the entire tree by two transactions, and that would prevent us from modifying anything in this sub-tree. The only lock compatible with SIX is IS.

Here is the compatibility matrix below; interpret the axes as being transaction $T1$ and transaction $T2$. As an example, consider the entry X, S – this means that it is not possible for $T1$ to hold an X lock on a resource while $T2$ holds an S lock on the same resource. NL stands for no lock.



Mode	NL	IS	IX	S	SIX	X
NL	Yes	Yes	Yes	Yes	Yes	Yes
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	Yes	No	No	No
S	Yes	Yes	No	Yes	No	No
SIX	Yes	Yes	No	No	No	No
X	Yes	No	No	No	No	No

7.1 Multiple Granularity Locking Protocol

1. Each Xact starts from the root of the hierarchy.
2. To get S or IS lock on a node, must hold IS or IX on parent node.
3. To get X or IX on a node, must hold IX or SIX on parent node.
4. Must release locks in bottom-up order.
5. 2-phase and lock compatibility matrix rules enforced as well
6. Protocol is correct in that it is equivalent to directly setting locks at leaf levels of the hierarchy.

Appendix

We now provide a formal proof for why the presence of a cycle in the waits-for graph is equivalent to the presence of a deadlock.

We use $\alpha_j(R_i)$ to represent the lock *request* of lock type α_j on the resource R_i by transaction T_j.

We use $\beta_{ij}(R_i)$ to represent a lock *held* of the lock type β_{ij} on the resource R_i by transaction T_j.

Definition 1. Deadlock

A deadlock is a sequence of transactions (with no repetitions) $T_1, \dots, T_k$ such that:

- for each $i \in [1, k)$, T_i is requesting a lock $\alpha_i(R_i)$, T_{i+1} holds the lock $\beta_{i,i+1}(R_i)$, and α_i and $\beta_{i,i+1}$ are incompatible, and
- T_k is requesting a lock $\alpha_k(R_k)$, T_1 holds the lock $\beta_{k,1}(R_k)$, and α_k and $\beta_{k,1}$ are incompatible.

Definition 2. Waits-for Graph

Let $T = \{T_1, \dots, T_n\}$ be the set of transactions and let $D_i \subseteq T$ be defined as follows:

- if T_i is blocked while requesting some lock $\alpha_i(R_i)$, then D_i is the set of transactions T_j that hold locks $\beta_{ij}(R_i)$ where α_i and β_{ij} are incompatible,
- otherwise, $D_i = \emptyset$.

The waits-for graph is the directed graph $G = (V, E)$ with $V = \{1, \dots, n\}$ and $E = \{(i, j) : T_j \in D_i\}$.

Theorem. There is a simple cycle in the waits-for graph $G \iff$ there is a deadlock.

Proof. Assume there is a simple cycle $C = \{(i_1, i_2), \dots, (i_{k-1}, i_k), (i_k, i_1)\} \subseteq E$.

By definition of the waits-for graph, $(i, j) \in E \iff T_j \in D_i$, or alternatively, that T_j holds a lock $\beta_{ij}(R_i)$ while T_i is blocked requesting $\alpha_i(R_i)$, and α_i and β_{ij} are incompatible.

Therefore, $(i_j, i_{j+1}) \in C \subseteq E \iff T_{i_{j+1}}$ holds a lock $\beta_{i_j i_{j+1}}(R_{i_j})$ while T_{i_j} is blocked requesting $\alpha_{i_j}(R_{i_j})$, where α_{i_j} and $\beta_{i_j i_{j+1}}$ are incompatible. A similar result holds for (i_k, i_1).

But this is simply the definition of a deadlock on the transactions $T_{i_1}, \dots, T_{i_k}$, so we have our result. □

Queue Skipping

An example of queue skipping is the following: Suppose, on resource A, that T_1 holds IS and T_2 holds an IX lock. The queue has, in order, the following requests: $T_3 : X(A)$, $T_4 : S(A)$, $T_5 : S(A)$, and $T_6 : SIX(A)$.

Now, let T_2 release its lock. Instead of processing the queue in order and stopping when a conflicting lock is requested (which would result in no locks being granted, as T_3 is at the front and wants $X(A)$), queue skipping processes the queue in order, *granting locks one by one whenever compatible.*

Here, it would look at T_3's X(A) request, determine that X(A) is incompatible with the IS(A) lock T_1 holds, and move to the next element in the queue. It would then grant T_4's S(A) request, as it is compatible with the held locks of A, and add $T_4 : S(A)$ to the set of locks held on A. It would then look at $T_5 : S(A)$, determine that it is compatible with $T_4 : S(A)$ and $T_1 : IS(A)$, and grant it. Finally, it would look at $T_6 : SIX(A)$, see that it is incompatible with $T_4 : S(A)$ and $T_5 : S(A)$ in the held set, and *not* grant it as a result.

Here is some pseudocode for processing the queue, but this time with queue skipping:

```
# If queue skipping is allowed, here is how to process the queue
H = set of held locks on A
Q = queue of lock requests for A

def request(lock_request):
    if lock_request is compatible with all locks in H:
        grant(lock_request)
    else:
        addToQueue(lock_request)

def release_procedure(lock_to_release):
    release(lock_to_release)
    for lock_request in Q:        # iterate through the lock requests in order
        if lock_request is compatible with all locks in H:
            grant(lock_request)   # grant the lock, updating the held set
```