Lecture 15: March 31

Lecturer: Prashant Shenoy *Scribe:* ***Nishi Poddar (2025), James Bardowski (2024)***

15.1 Overview

This lecture covers the following topics:

Distributed Transactions: ACID properties, Transaction Primitives, Private Workspace, Write-ahead logs.

Concurrency control and locks: Serializability, Interleaving, Optimistic Concurrency Control, Two-phase Locking (2PL), Timestamp-based Concurrency Control.

15.2 Transactions

Transactions provide a higher mechanism for atomicity of processing in distributed systems. *Atomicity* is when a set of operations is protected with the all or nothing property, i.e., either all of the operations succeed or none of them succeed. Anything that is protected by a transaction operates as one atomic operation even though there may be multiple statements.

Let us try and understand Transactions and their importance using an example. Let us assume there are two clients, client 1 and client 2, which are trying to make a transaction on bank accounts A, B, C. Let's further assume accounts A, B, C has $100, $200, $300 respectively initially. Client 1 wants to transfer $4 from account A to account B and client 2 wants to transfer $3 from account C to account B. In the end, $7 needs to be deposited into account B from client 1 and client 2. To transfer, client 1 needs to read and deduct balance in account A and then transfer by reading and updating balance in account B. Similarly, client 2 needs to read and deduct balance in account C and then transfer by reading and updating balance in account B.

Let us say if client 1 and client 2 makes RPC calls to bank's database to perform their respective operations at the same time. There are many possible ways all of the operations from client 1 and client 2 could be interleaved. Figure 15.1 shows one possible interleaving.

Clients are executing parallel queries, so can't assume anything about the order in which the operations take place. Initially client 1 reads, deducts and updates the balance in A. Next, client 2 reads, deducts and updates the balance in C. In the next step, client 1 reads balance in account B and then client 2 reads and add $3 to the balance in account B. But client 1 still has the old value and it adds $4 to old value and updates the balance in account B by overwriting the changes made by client 2. In the end, only $4 were transferred to account B instead of $7. This interleaving gave us incorrect result.

Figure 15.2 shows how you want the operations to happen. All of the operations by a particular client happen like one atomic operation. The order of which client executes first does not matter.

One way to achieve this would be to use a lock on the entire database, so only one of client 1 or 2 can access the database at a time. The issue with that is it would make all the operations on the database

Client 1	Client 2
Read A: $100	
Write A: $96	
	Read C: $300
	Write C:$297
Read B: $200	
	Read B: $200
	Write B:$203
Write B:$204	

Figure 15.1: Depiction of sequence of transactions by two clients

Client 1	Client 2
Read A: $100	
Write A: $96	
Read B: $200	
Write B:$204	
	Read C: $300
	Write C:$297
	Read B: $204
	Write B:$207

Figure 15.2: Atomic transactions.

sequential (i.e. only one client can write to it at a time) but you may have multiple clients sending requests at the same time, so the performance would degrade significantly. So what we want is for the database to physically execute concurrently, while logically it seems like it is executing sequentially, which is achieved through transactions. Transactions essentially allow you to take locks on a set of operations.

15.2.1 ACID Properties

- *Atomic:* All or nothing. Either all operations succeed or nothing succeeds.
- *Consistent: Consistency* is when each transaction takes system from one consistent state to another. A *consistent state* is a state where everything is correct. After a transaction, the system is still consistent.
- *Isolated:* A transaction's changes are not immediately visible to others but once they are visible, they become visible to the whole world. This is also called the *serializable* property. This property says

that even if multiple transactions are interleaved, the end result should be same as if one transaction occurred after another in a serial manner.

- *Durable:* Once a transaction succeeds or commits, the changes are permanent, but until the transaction commits, all of the changes made can be reverted.

Question: If there is a rollback in a transaction, at what point can it happen?
Answer: In the example above, client A has four instructions which are together enclosed in a transaction. You can decide to abort a transaction at any point during the execution, in which case the amounts will revert to the original value. But once you have committed the transaction then there is no going back and all the changes become visible.

Question: How are multiple requests (i.e. multiple clients) executing together?
Answer: This will be achieved by something called concurrency control, which we will get to. In brief, we have to implement finer grain locks- not locks on the whole database but locks on individual records.

15.2.2 Transaction Primitives

Special primitives are required for programming using transactions. Primitives are supplied by the operating system or by the language runtime system.

- BEGIN_TRANSACTION : Marks the start of transaction.
- END_TRANSACTION : Terminate the transaction and try to commit. Everything between begin and end primitive will be executed as one atomic set of instructions.
- ABORT_TRANSACTION : Kill the transaction and undo all of the changes made.
- READ : Read data from a file, a table, or otherwise.
- WRITE : Write data to a file, a table, or otherwise.

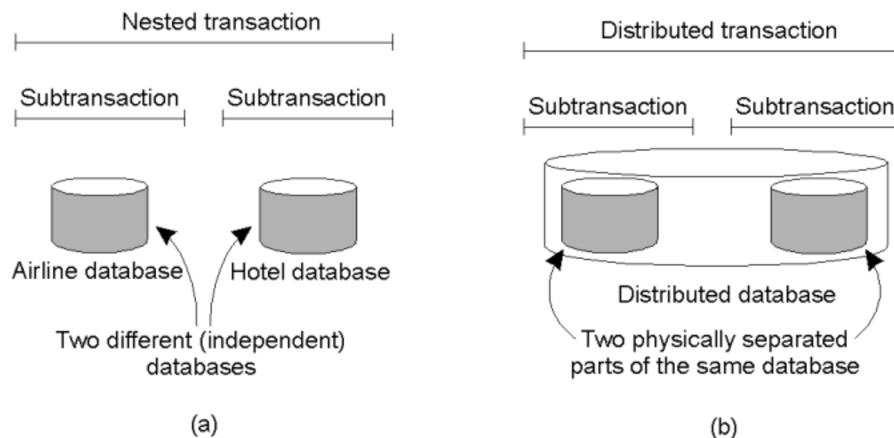

Figure 15.3: Nested transactions and distributed transactions.

15.2.3 Distributed Transactions

Two concepts: nested and distributed transactions

a) *Nested Transaction*

Here one transaction is nested inside another. That is, within the BEGIN and END block that denotes one transaction there is another BEGIN and END block.

Take an example of making a reservation for a trip which includes airline and hotel reservations. Assume you want to either do both flight and hotel booking or neither. Usually airlines and hotels are different companies and have their own databases. This can be achieved using *nested transactions*. This way, if one booking fails, you undo the changes made for other booking. So, the smaller transactions protects each booking and the bigger transaction gives ACID properties as a whole. If any one small transaction fails, the complete transaction is aborted.

b) *Distributed Transaction*

A transaction is distributed if the operations are being performed on data that is spread across a database that is distributed (i.e. not on one machine). From user's perspective, there is only one logical database. So, to make a transaction on this logical database, we will have subtransactions. Each subtransaction perform operations on a different machine. Performing operations on distributed database needs distributed lock which makes implementation difficult.

Transactions are not database specific, even though we talk about them in that context. We can have transactions in a distributed web app if we want.

Question: Is it possible that one transaction is successful in one database and fails in a mirror database?
Answer: A distributed database is one in which data are partitioned into multiple databases instead of one database being a mirror (exact copy) of the other.

15.2.4 Implementation

We will see two ways to implement distributed transactions- private work spaces and write-ahead logs. Both these methods work for both single transaction systems as well as distributed transaction systems.

Private Workspace

- Every transaction gets its own copy of the database to prevent one transaction from overwriting another transaction's changes. Instead of a real copy, each transaction is given a snapshot of the database, which is more efficient. Each transaction makes changes only to its copy and when it commits, all of the updates are applied to database.
- Making a copy is optimized by using copy-on-write. A copy is not made for read operations.
- Using a copy also makes aborting a transaction easy. If a transaction is aborted, the copy is simply deleted and no changes are applied to the original database. If a transaction is committed, you just take the changes and apply them to the database.

In the above figure, the index is used to store the locations of the file blocks. To execute a transaction, instead of making copies of the file blocks, a copy of the index is made. The index initially points to original file blocks. For a transaction, it looks like it has its own copy. When the transaction needs to make an

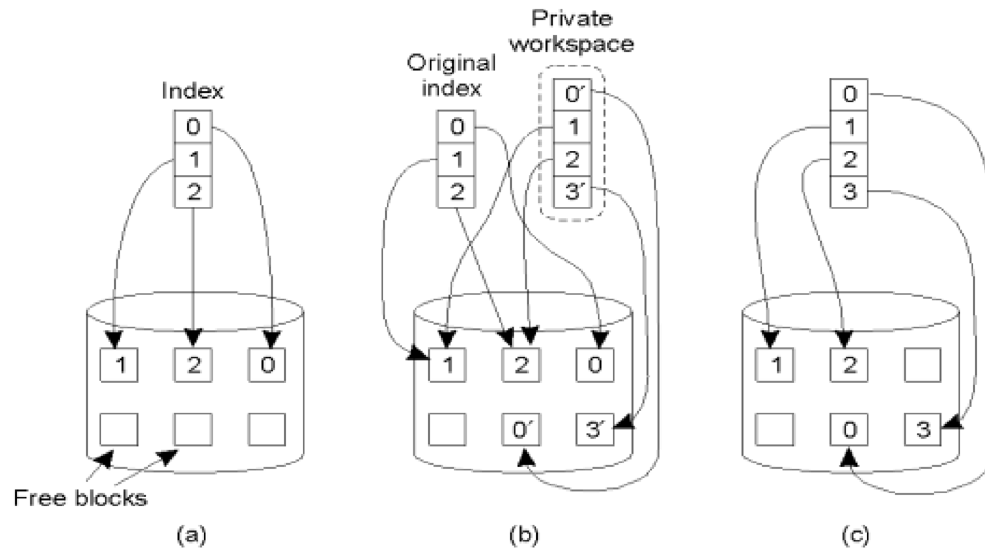

Figure 15.4: Private workspaces.

update to block 0, instead of making change to original block, a copy of the block (0′) is created and the change is made to the copy. Now the transaction index is made to point to the copy instead of original. In case the transaction adds something to the database, a new free block 3′ is created. Essentially, we are optimizing by making a copy only when a write operation is executed. If the transaction is aborted, the copies are deleted. If the transaction is committed, the changes made are applied to the original database. This is the *private workspace model*. Downside: making a copy of a large DB is not cheap; a lot of work if there are small changes. So, we need very efficient way for making copies. A virtual copy will allow for efficient copies – use copy on write. A private workspace is a copy of the entire DB.

Question: Would you need to copy the entire index? Is that not inefficient?
Answer: Yes, we copy the whole index and it is much smaller than the actual data.

Question: At any given point in time is there only one transaction being processed?
Answer: That is not the case, we want an arbitrary number of transactions to execute in parallel. Each transaction will have its own private workspace- for N transactions we will have N workspaces (and the original database) where they make their changes.

Question: If two transactions make copies to the same block at the same time what do you do?
Answer: That is called the write-write conflict. Either you have to abort both and restart them or let one of them succeed. In private workspaces, aborting the workspace is as simple as just discarding the private workspace index copy that was created. In a transactional system, whenever there are conflicts the transaction will abort.

Question: Will the index be stored on a single machine or distributed machines?
Answer: At this point we are just looking at a logical view of what is happening. The notion of private workspace works well if you have a single machine and database or multiple machines and databases. If you have many machines then you have private workspaces that span multiple disks, but the concepts remain the same- you use copy and write, with efficient snapshots, and commit if there are no conflicts or discard if there are.

Question: Is this used in real systems, since there are accounts being changed that will cause issues if they are discarded?

Answer: We will talk about concurrency control, which is a process by which you let multiple transactions execute in parallel and yet get safe results. A version of that is called optimistic concurrency control where you don't use locks, allow transactions to make changes, and at the point where they are about to commit check if two transactions have overwritten each other. If they have we declare a write-write conflict and abort. Most transactions are not modifying the same piece of data, so with high probability will successfully commit. In the case where you do have multiple transactions frequently modifying the same piece of data then optimistic concurrency control will lead to many abortions, so we will instead favor pessimistic concurrency control.

Question: When do you merge the change? Can you merge it while another transaction is ongoing
Answer: Yes, there can be an arbitrary number of transactions ongoing. The only thing we need to know is when we started executing did someone else also modify the same data items that you did, in which case you abort.

Question: If two concurrent transactions make changes to their own copy of the same block and if the first transaction commits, does the second transaction overwrite the changes made by the first transaction?
Answer: If a transaction wants to commit and meanwhile some changes were made to the same block the transaction wants to commit to, this is a write-write conflict. In this case, the transaction is aborted.

Write-ahead Logs

In this design, the transaction make changes to the live database instead of a copy. We instead keep a transaction log in separate file (called a write-ahead log) to note the changes the transaction is making to the database. Here committing is trivial since the changes are already executed on the live database, but aborts are harder because we will have to undo the changes by scanning the write-ahead log and undoing each operation listed to restore the database to where it was before the transaction. The undo process is called a *rollback*.

	Log	Log	Log
x = 0;			
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

- a) A transaction
- b) – d) The log before each statement is executed

Figure 15.5: Distributed transactions

The above figure shows a trivial transaction. The database has only two entries, x and y, initialized to 0. The transaction has three operations as shown in the figure. After the first operation, and entry with the original and updated values of x is added to the log. After the second operation, another entry storing the

original and new value of y is added. In the last step, another new entry storing the previous and updated value of x is added to the log. If the transaction commits, there is nothing to do as the changes were made to original database. A commit success log is added to undo log in the end. To undo the changes, the log is traversed backwards reverting each operation by replacing current value with the previous value.

Question: How long do we maintain the logs?
Answer: The logs are maintained in files in the database. Once a log file grows to a certain size, you switch to a new file and eventually sometimes discard the old files.

Question: Are x and y here the indices of the data?
Answer: No, x and y are the actual data values in the database.

Question: How does the transaction know if it's aborted?
Answer: Two ways to abort: explicit statements that the user wrote, or locking as we will see. The lock manager will abort if needed.

Question: If we're already making changes to the database and another transaction comes and modifies the data how will it be handled?
Answer: When it is time to commit, we have to see if anyone else has modified the changes we did, in which case we flag a conflict and undo the changes. The transaction system can only do one of two things- either as you make the changes you take locks from x and y so that no other transaction can modify it (pessimistic case), or if we don't hold locks while making changes, we check at commit time if anyone else made changes, and if so we abort the same thing (optimistic case).

Question: Is the log per transaction or is there one log shared by all?
Answer: The transaction log is a single file. Multiple transactions are going to write to that file, what isn't shown here is that each of these entries has to have a transaction ID associated with it, so that when we are scanning the log back we just look for that ID in order to revert changes.

Question: When you roll back, do you have to scrape out the IDs?
Answer: You don't have to change the log. The log will have aborted transaction, you just have to undo what is in the database. You don't have to delete from the log.

Question: What is the case where the optimistic approach is better?
Answer: The idea behind optimistic approach is that for a large database with millions of records, and any given transaction will only make changes to a small part of the database. Thus, even if there are many transactions, if each work on different parts of the database they will rarely conflict.

Question: If C is about to commit and it sees that B has made changes, so you go to abort C, and when B is about to commit you see it has caused problems with A, how does the abort take place?
Answer: Here you have a case called the cascading rollback, where each abort causes a problem with another transaction due to a write-write conflict, so both have to termintated, which conflicts with another one and so on. That is possible, and is a bad case because multiple transactions all get rolled back at the same time. The only solution here, to avoid many conflicts, is to take locks.

Question: What does "force logs on commit" mean?
Answer: If transaction is committed, a entry is added to the log to indicate that the transaction is successful and there is no need for undo.

Question: What happens if multiple transactions are operating on x,y in the above example?
Answer: One approach is to use locks. While one transaction is operating, it holds a lock which prevents any other transaction from making any changes. Another approach is *optimistic concurrency control.* This approach does not use locks and assumes that transaction conflicts are rare. Conflicts are tracked and all of the transactions that are part of the conflict are aborted and restarted again.

15.3 Concurrency Control

The goal of concurrency control is to allow several transactions to execute in parallel but make it seem serialized. We want to avoid each transaction taking a lock on the entire database because it would lead to poor performance if many users want to execute transactions at the same time. In other words, we don't want sequential execution (wait times would balloon during heavy loads), which is what global locking will give us and won't allow concurrency. Instead, we want all the transactions to execute in parallel while still achieving consistency. If all the transactions commit successfully, the final result should appear as if they executed sequentially.

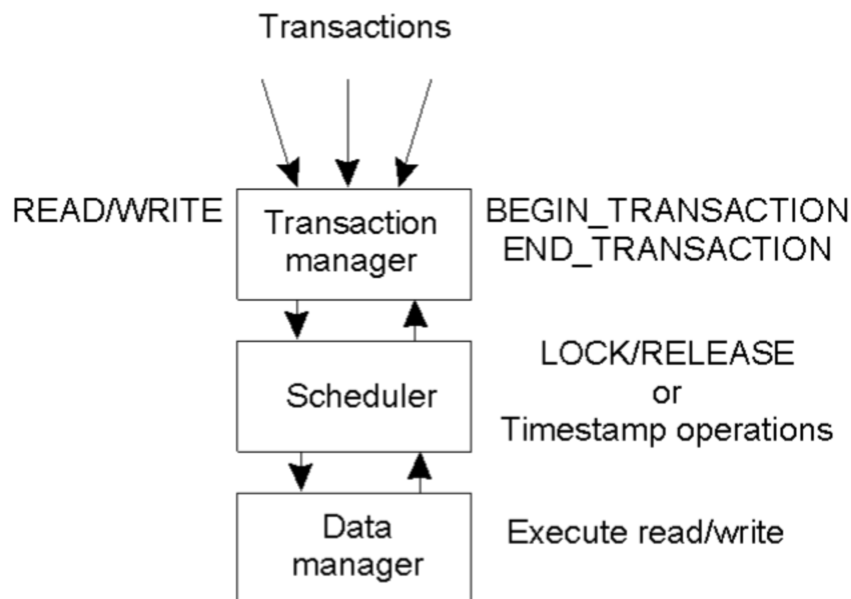

Figure 15.6: General organization of managers for handling transactions.

Concurrency can be implemented in a layered fashion as shown in the prior figure. The transaction manager implements the private-workspace model or write-ahead log model. The scheduler implements locking and releasing the data (in this case, we are taking locks of records of the database rather than the entire database). The data manager makes changes to either the workspace (index) or to the actual database.

In the case of distributed systems, a similar organization of managers is applied as shown in the next figure. Data is now split across multiple machines. A scheduler is on each machine and needs to handle distributed locking now. Beyond that everything is same. Question is what should be in the scheduler for us to get concurrency? We will look at that next.

Question: How do you deal with databases that are distributed or replicated?
Answer: The transaction manager sits at the top. If there are 3 copies of data, it acquires 3 locks and ensures that any changes are propagated to all copies.

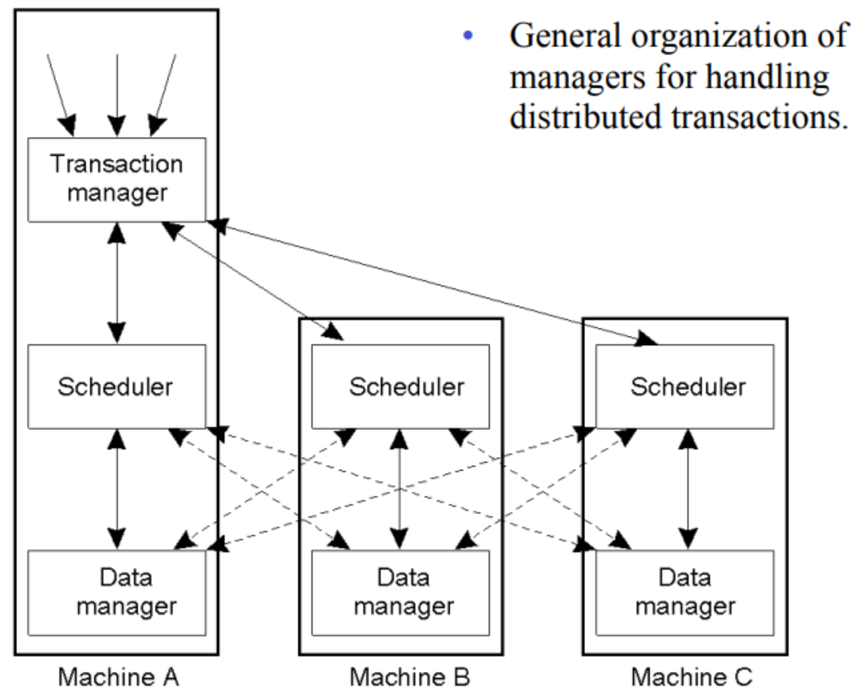

Figure 15.7: General organization of managers for handling distributed transactions.

15.3.1 Serializability

This is the key property imposed on the end result of a transaction: idea is to properly schedule conflicting operations. The end result of concurrent transactions should be to look as if the transactions were executed serially. The next figure shows an example where each transaction modifies x, and three possible ways the transactions are interleaved. The result is valid only if it is same as the result of one possible serial orders (1,2,3 or 3,2,1 or 2,3,1 etc). If the six operations associated with the three transactions (two in each transaction) execute in parallel, we can obtain any arbitrary interleaving of the six operations. We then check if there is a sequential execution of transactions that could have produced such an interleaving, in which case it is considered valid. The last transaction will set the final state of 'x'.

In the above example, Schedule 1 is valid because the output is same as if the transactions are executed in the a,b,c serial order. In Schedule 2, we see some interleaving of instructions- x=0 is executed twice before x=1 and x=2. Here we check the end state, which is that x = 3, and check if there is some sequential order that would lead to the same end state. Since there is such an order (the same a,b,c order as Schedule 1), we say that Schedule 2 is also legal. Schedule 3 is illegal, however, because the end state interleaves x as value 5, and there is no sequential execution that could achieve the same end state.

Question: Is the scheduler actually going to simulate this entire process?
Answer: The scheduler is not going to do that. This is just a concept called serializability- if you have operations executing concurrently, the output of these operations should be as if they were executed in some sequential order. If you want to implement serializability, it is not the scheduler who should figure this out, you would rather have some protocol do this automatically.

Interleaving could result in two kinds of conflicts: read-write conflicts and write-write conflicts. Read-write

```
BEGIN_TRANSACTION
 x = 0;
 x = x + 1;
END_TRANSACTION
```
(a)

```
BEGIN_TRANSACTION
 x = 0;
 x = x + 2;
END_TRANSACTION
```
(b)

```
BEGIN_TRANSACTION
 x = 0;
 x = x + 3;
END_TRANSACTION
```
(c)

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	Illegal

Figure 15.8: Example of Serializable and Non-Serializable transactions.

conflicts occurs when the transaction reads, performs a write based on the value of the read data, but sees that the data has been updated since the read took place. Write-write conflicts occur if one write operation overwrites another write operation's update. The scheduler should acquire the appropriate locks to prevent both of the conflicts from happening.

15.3.2 Optimistic Concurrency Control

In the *optimistic concurrency control* technique, there are no locks or lock manager. The transaction is executed normally without imposing serializability restriction, but the transaction is validated at the end by checking for read-write and write-write conflicts. If any conflict is found, all of the transactions involving in the conflict are aborted. This design takes an optimistic view and assumes database is large and most of the transactions occur on different parts of the database. Using locks adds unnecessary overhead and this design avoids this by checking for validity in the end. It works well with private workspaces because the copies can be deleted easily if a transaction aborts.

Advantages:

- One advantage is that this method is deadlock free since no locks are used.
- Since no locks are used, this method also gives maximum parallelism.

Disadvantages:

- The transaction needs to be re-executed if it is aborted.
- The probability of conflict rises substantially at high loads because there are many transactions operating at the same time and probability of them operating on same data block is high. Throughput will go down when the load is high.

While there are some distributed services that use this approach, it is not widely used, especially by commercial databases because of high abort rates at high loads.

Question: If there are no locks, how can you track if the same data was accessed?
Answer: Every transaction has its own copy of the data and its previous values and timestamps to verify changes. There are many ways to do this.

Question: If two transactions that are conflicted are aborted and re-executed again, will it not result in conflict again?
Answer: If they are executed at the same time again, they will conflict. The scheduler needs to randomize execution time or something else so that they are executed at different times and conflict is avoided.

15.3.3 Two-phase Locking (2PL)

Pessimistic concurrency control requires the use of locks. One widely used protocol for this is called two-phase Locking (2PL), which is a standard approach used in databases and distributed systems. The scheduler grabs locks on all of the data items the transaction touches and is released at the end when the transaction ends. If a transaction touches an item and lock is grabbed, no other transaction can touch that data item. The transaction needs to wait for lock to be released if it wants to operate on locked data.

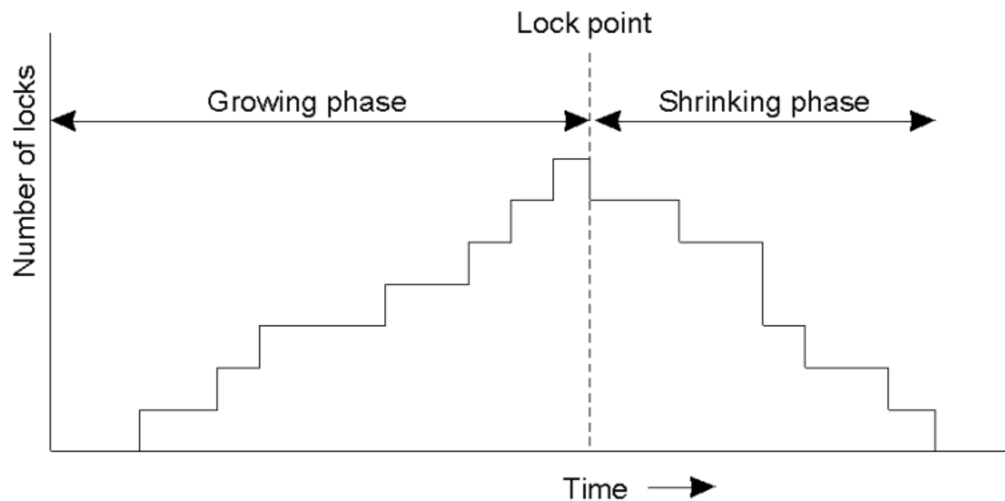

Figure 15.9: Two-Phase Locking

Additionally, a constraint is imposed that if a transaction starts releasing locks, it cannot acquire it again. This leads to two phases in each transaction as shown in the above figure. During the growing phase, the transaction acquires locks and once it releases a lock, the transaction cannot acquire any more locks. This is shrinking phase as the number of locks the transaction is holding reduces. The transaction needs to make sure that it will not touch any new data before releasing first lock as it cannot acquire a lock again.

A simpler approach is to completely avoid shrinking phase. As shown in the last figure, Strict Two-Phase Locking grabs locks and releases all of the locks at the same time before committing. In this method, the transaction will hold the lock until the end and does not release immediately as soon as it is done with the lock. One of the disadvantages of these approaches is that they are prone to deadlocks and additional constraints need to be implemented to avoid deadlocks.

Question: Where is this locking mechanism implemented? Since the lock acquisition and release happens in the transaction, shouldn't it be the responsibility of the transaction manager?
Answer: The locking mechanism is implemented in the scheduler. Essentially, the transaction manager

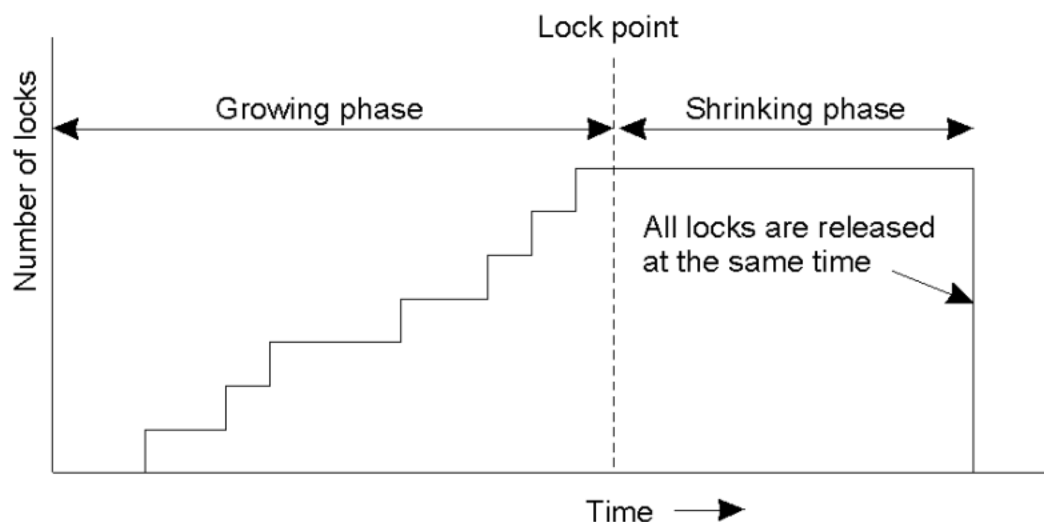

Figure 15.10: Strict Two-Phase Locking

itself notifies scheduler to grab the lock on a specific data block. The transaction manager works with the scheduler to grab the locks.

15.3.4 Timestamp-based Concurrency Control

This method handles concurrency using timestamps (not locks), in particular Lamport's clock (logical clock instead of physical clock). The timestamp is used to decide the order and which transaction to abort in case of read-write or write-write conflicts. If two transactions are conflicted, the earlier transaction should be aborted and the transaction that started later should be allowed to continue. For each data item x, two timestamps are tracked:

- Max-rts(x): max time stamp of a transaction that read x.
- Max-wts(x): max time stamp of a transaction that wrote x.

Conflicts are handled using both these timestamps as shown in the Figure 16.13. If a transaction want to perform read opearion on data item x, the last write on that data item is checked. The transaction timestamp is compared with the last write timestamp of the data. If the later transaction modified the data, the transaction is aborted. If the read is successful, the read timestamp is updated by calculating max of the previous timestamp and timestamp of current transaction as shown in the above figure.

In case of a write, if there is any more recent transaction that has read or modified the data item, the transaction is aborted. If the write is performed, the timestamp of data item is updated.

Question: When you undo the transaction, do you undo the changes?
Answer: During abort, the state is restored to the original values using the undo log in case of Write-ahead log and copies are deleted in private workspace model.

Question: How do we ensure the atomicity of read, write operations and the checks made in Figure 16.13?
Answer: A lock is grabbed while performing all of the operations shown in the figure to prevent any other transaction from making changes.

- *$Read_i(x)$*
  - If $ts(T_i) < max\text{-}wts(x)$ then Abort T_i
  - Else
    - Perform $R_i(x)$
    - $Max\text{-}rts(x) = \max(max\text{-}rts(x), ts(T_i))$
- *$Write_i(x)$*
  - If $ts(T_i) < max\text{-}rts(x)$ or $ts(T_i) < max\text{-}wts(x)$ then Abort T_i
  - Else
    - Perform $W_i(x)$
    - $Max\text{-}wts(x) = ts(T_i)$

Figure 15.11: Read-writes using timestamps

Question: When a transaction is aborted, is it just killed or rerun again?
Answer: There are two ways to handle this. One way is to inform the application that the transaction is aborted and let the application rerun the transaction again. Another way is to make the transaction manager retry the transaction.