CMPSCI 677 Operating Systems **Spring 2022**

Lecture 15: March 23

Lecturer: Prashant Shenoy *Scribe:* ***Dinesh Kandula***

15.1 Overview

This lecture covers the following topics:

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

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

15.2 Transactions

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. 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. Let's assume accounts A, B, C has $100, $200, $300 respectively initially. 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. As shown in Figure 15.1, 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.

An important thing to note is that this database is shared by multiple clients. To solve the above issue, you could try fine-grained locking by locking the account before performing a read or write and then release the lock once the operation is done. This is still not going to solve the issue as the same interleaving can still happen. Instead, by locking the entire database and performing all of the operations each client needs to do before releasing the lock, this issue can be solved. But locking entire database will prevent other clients from performing any operations while one client is executing. This will degrade the performance significantly.

Question: If you lock the entire object and perform all of the operations on that object before releasing it, could this solve the above issue instead of locking the entire database?
Answer: Locking an object across multiple operations is not fine-grained locking. Fine-grained locking, which is locking and releasing for each operation, cannot the solve the above issue. Coarser-grained locking

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

which involves locking the database or row across multiple operations can solve the issue but can degrade the performance.

Transactions are the higher level mechanism which provides atomicity of processing. *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.

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.

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.

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.

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.

15.2.3 Distributed Transactions

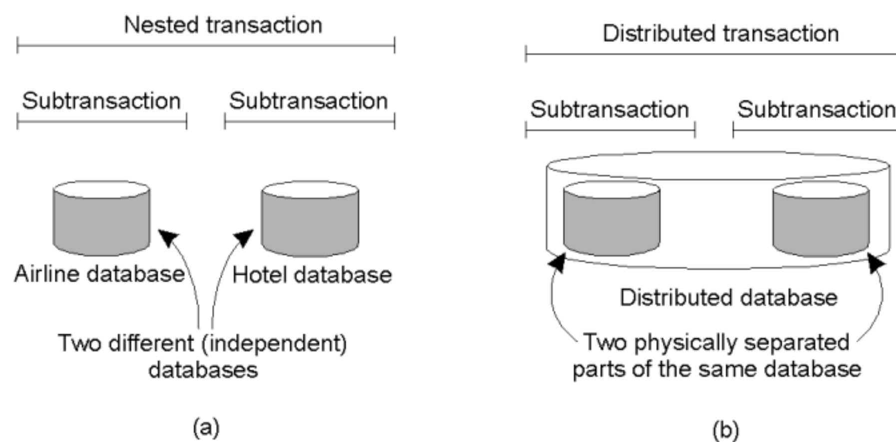

Figure 15.3: Distributed transactions.

a) *Nested Transaction*

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*. There are smaller transactions inside the main BEGIN_TRANSACTION and END_TRANSACTION primitives. 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 two or more databases. 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.

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

Private Workspace

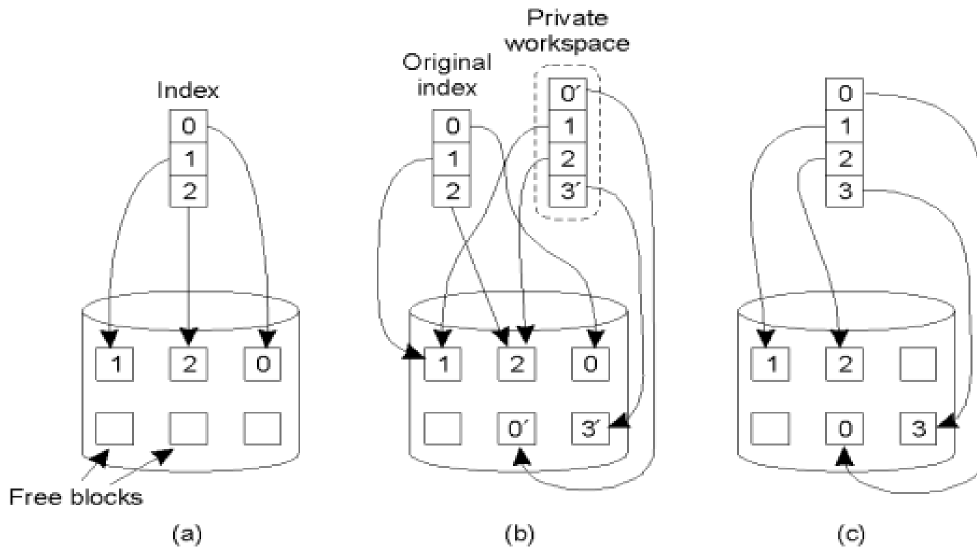

Figure 15.4: Distributed transactions.

- 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 Figure 15.4, 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 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*.

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 actual database. If the transaction aborts, all of the changes needs to be aborted. The value before the change is stored using an undo log. So, when the transaction aborts, the undo log is used to rollback and restore the original value. Commits need to be atomic to avoid write-write conflicts.

Figure 15.5: Distributed transactions

Figure 15.5 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, the original value of x and the update value 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, again a new log 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 abort, the log is traversed backwards reverting each operation by replacing current value with the previous value.

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

To handle concurrency, a trivial solution is to use a lock over the complete database and allow only one transaction at a time. This solution will lead to very bad performance as only one transaction is executed at a time. So, several transactions should be allowed to be execute simultaneously for better performance. The final result after concurrent transactions should be same as if each transaction ran sequentially.

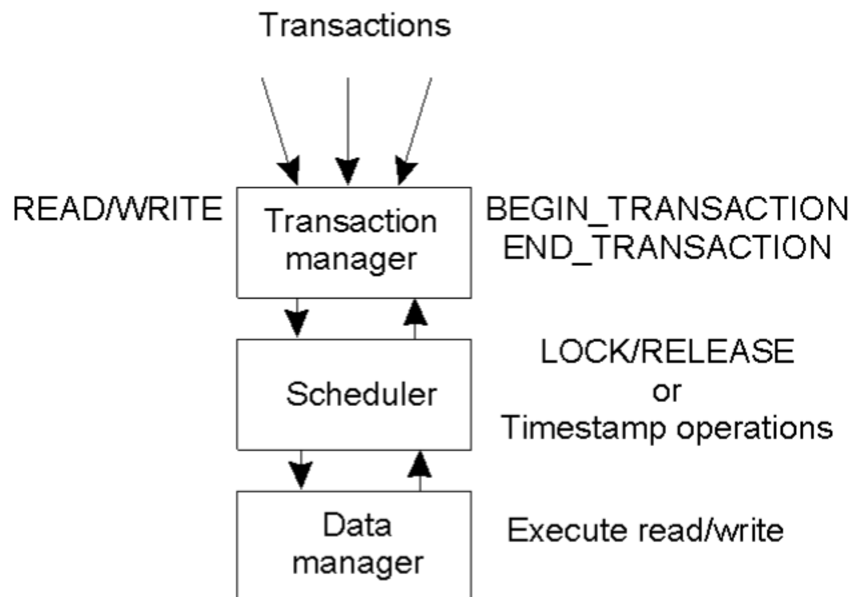

Figure 15.6: General organization of managers for handling transactions.

Concurrency can be implemented in a layered fashion as shown in Figure 15.6. The transaction manager implements the private-workspace model or write-ahead log model. The scheduler implements locking and releasing the data. 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 Figure 15.7. Data is now split across multiple machines. The scheduler needs to handle distributed locking now. Beyond that everything is same.

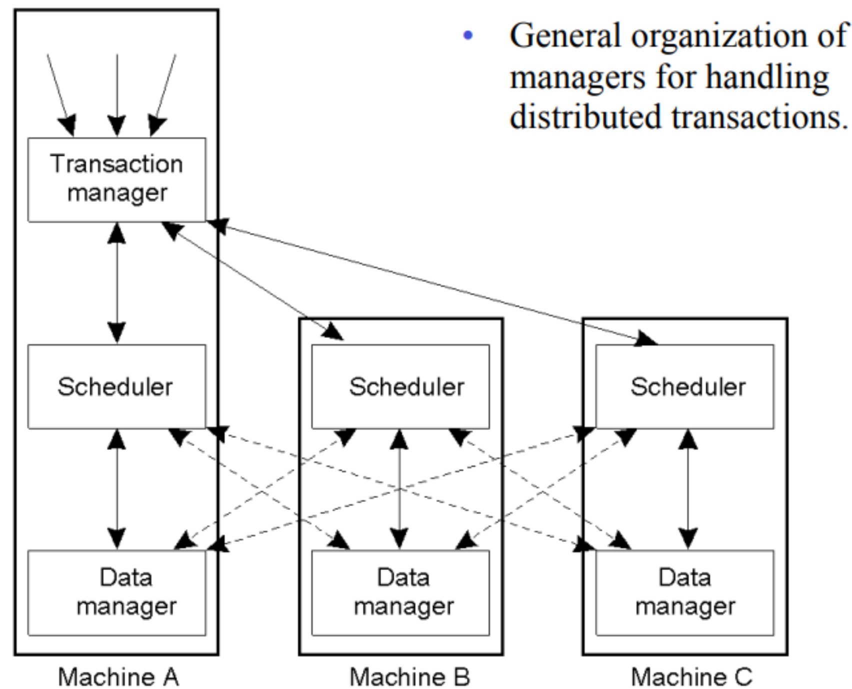

Figure 15.7: General organization of managers for handling distributed transactions.

15.3.1 Serializability

This the key property that is imposed on the end result of a transaction. The end result of a concurrent transactions should be same as if the transactions are executed serially. Figure 15.8 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 a sequential order can be found that gives the same result as the concurrent transactions, then that interleavability is considered valid.

In the example shown in Figure 15.8, Schedule 1 is valid because the output is same as if the transactions are executed in the a,b,c serial order. Schedule 2 is also valid because result is same as the result of a,b,c serial order. Schedule 3 is illegal because there is no possible sequential execution of a,b,c that gives the same result as Schedule 3. Interleaving could result in two kinds of conflicts: read-write conflicts and write-write conflicts. Read-write conflicts occurs when the transaction reads an outdated value of a variable and update it based on the outdated value. 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 *optimistic concurrency control*, 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

```
BEGIN_TRANSACTION          BEGIN_TRANSACTION          BEGIN_TRANSACTION
  x = 0;                     x = 0;                     x = 0;
  x = x + 1;                 x = x + 2;                 x = x + 3;
END_TRANSACTION            END_TRANSACTION            END_TRANSACTION

      (a)                        (b)                        (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.

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.

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)

Because the probability of conflicts is high at high loads, pptimistic concurrency control is not widely used in commercial systems. Two-phase Locking (2PL) 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.

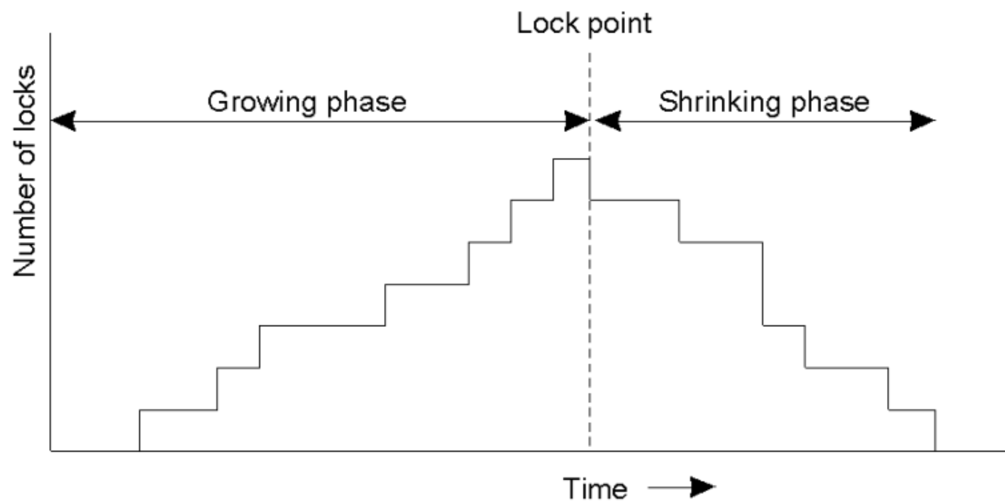

Figure 15.9: Two-Phase Locking

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 Figure 15.9. 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.

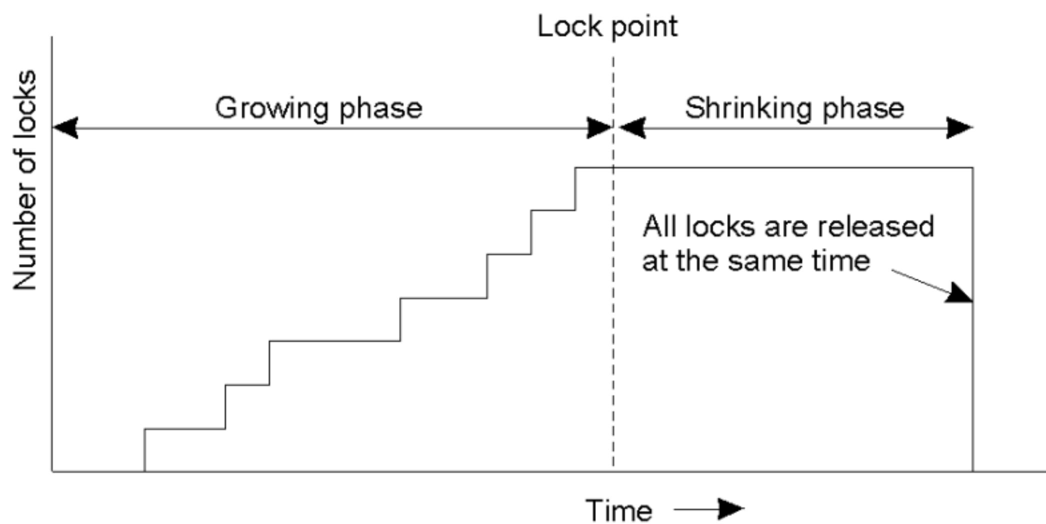

Figure 15.10: Strict Two-Phase Locking

A simpler approach is to completely avoid shrinking phase. As shown in Figure 15.10, Strict Two-Phase Locking grabs locks and releases all of the locks at a 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.

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 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, in particular Lamport's clock (logical clock instead of physica 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 later transaction should be aborted and the transaction that started early 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.

- $Read_i(x)$
  - 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

Conflicts are handled using both these timestamps as shown in the Figure 15.11. 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 15.11?
Answer: A lock is grabbed while performing all of the operations shown in the figure to prevent any other transaction from making changes.

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.