UNIT – IV

(Distributed Transactions)

Dr. K. Jairam Naik

NIT Raipur

ATOMIC TRANSACTIONS

- An *Atomic Transaction* (or just transaction for short) is **a computation consisting of a collection of operations** that take place indivisibly in the presence of failures and concurrent computations.
- That is, **either all of the operations are performed successfully or none** of their effects prevail, and *other processes executing concurrently cannot modify or observe* intermediate states of the computation.
- Transactions **help to preserve the consistency of a set of shared data objects** in the face of failures and concurrent access.
- **They make crash recovery much easier**, because a transaction can only end in *two states transaction carried out completely or transaction failed completely*.

Transactions have the following essential properties:

1. **Atomicity.** This property ensures that to the outside world *all the operations of a transaction appear to have been performed indivisibly*. Two essential requirements for atomicity are *atomicity with respect to failures* and *atomicity with respect to concurrent access*.

   i. ***Failure atomicity*** ensures that if a transaction's work is interrupted by a failure, any partially completed results will be undone. Failure atomicity is *also known as the all-or-nothing* property because a transaction is always performed either completely or not at all.

   ii. ***Concurrency atomicity*** ensures that while a transaction is in progress, *other processes executing concurrently with the transaction cannot modify or observe intermediate states* of the transaction.

   Only the final state becomes visible to other processes after the transaction completes. Concurrency atomicity is *also known as consistency property* because a transaction moves the system from one consistent state to another.

2. **Serializability.** This property (also known as *isolation property*) ensures that *concurrently executing transactions do not interfere with each other*. That is, the concurrent execution of a set of two or more transactions is serially equivalent.

3. **Permanence.** This property (also known as *durability property*) ensures that *once a transaction completes successfully, the results of its operations become permanent and cannot be lost* even if the corresponding process or the processor on which it is running crashes.

NOTE:

•The A, C, I, and D respectively stand for atomicity (failure atomicity), consistency (concurrency atomicity), isolation (serializability), and durability (permanence).

•Therefore, transaction properties are also referred to as ACID properties.

Need for Transactions In a File Service

1. *For improving the recoverability of files in the event of failures.*

   A file service that does not support transaction facility, unexpected failure of the client or server process during the processing of an operation may leave the files that were undergoing modification in an inconsistent state.

2. *For allowing the concurrent sharing of mutable files by multiple clients in a consistent manner.*

   Transaction facility is basically a high-level synchronization mechanism that properly serializes the access requests from multiple clients to maintain the shared file in the intended consistent state.

Inconsistency Due to System Failure

The following examples illustrate how the transaction facility of a file service helps to prevent file inconsistencies arising from events beyond a client's control, such as machine or communication failures or concurrent access to files by other clients.

a_1: read balance (x) of account X
a_2: read balance (z) of account Z
a_3: write ($x - 5$) to account X
a_4: write ($z + 5$) to account Z

Fig. 9.7 A set of operations to transfer \$5 from account X to account Z.

- Consider the banking transaction of *Figure 9.7*, which is comprised of four operations (a1, a2, a3, a4) for transferring $5 from account X to account Z.
- Suppose that the customer account records are stored in a file maintained by the file server.
- Read/write access to customer account records are done by sending access requests to the file server.
- In the base file service without transaction facility, the job of transferring $5 from account X to account Z will be performed by the execution of operations a1, a2, a3, and a4 in that order.
- Suppose the initial balance in both the accounts is $100.
- Therefore, if all the four operations are performed successfully, the final balances in accounts X and Z will be $95 and $105, respectively.

Let the initial balance in both the accounts of Figure 9.7 be $100.

<table>
<tr><td>Successful execution</td><td colspan="2">Unsuccessful execution</td></tr>
<tr><td>a_1: $x = 100$
a_2: $z = 100$
a_3: $x = 95$
a_4: $z = 105$</td><td colspan="2">a_1: $x = 100$
a_2: $z = 100$
a_3: $x = 95$
System crashes</td></tr>
<tr><td>Final result</td><td colspan="2">Final result</td></tr>
<tr><td rowspan="2">$x = 95$
$z = 105$</td><td>When the four operations are not treated as a transaction</td><td>When the four operations are treated as a transaction</td></tr>
<tr><td>$x = 95$
$z = 100$</td><td>$x = 100$
$z = 100$</td></tr>
</table>

Fig. 9.8 Possible final results in successful and unsuccessful executions with and without transaction facility of the operations of Figure 9.7.

Inconsistency Due to Concurrent Access

a_1: read balance (x) of account X
a_2: read balance (z) of account Z
a_3: write ($x - 5$) to account X
a_4: write ($z + 5$) to account Z

T_1 : Transfer \$5 from account X to account Z.

b_1: read balance (y) of account Y
b_2: read balance (z) of account Z
b_3: write ($y - 7$) to account Y
b_4: write ($z + 7$) to account Z

T_2 : Transfer \$7 from account Y to account Z.

Fig. 9.9 Two banking transactions.

- Consider the two banking transactions T1 and T2 of *Figure 9.9*.
- Transaction T1, which is meant for transferring \$5 from account X to account Z, consists of four operations a1, a2, a3' and a4.
- Similarly, transaction T2, which is meant for transferring \$7 from account Y to account Z, consists of four operations b1, b2 , b3, and b4.

- The net effects of executing the two transactions should be the following:
  - To decrease the balance in account X by \$5
  - To decrease the balance in account Y by \$7
  - To increase the balance in account Z by \$12

- Assuming that the initial balance in all the three accounts is $100, the final balances of accounts X, Y and Z after the execution of the two transactions should be $95, $93, and $112, respectively.

- However, notice that the complete serialization of all transactions (completing one before the next one is allowed to commence) that access the same data is unnecessarily restrictive and can produce long delays in the completion of tasks.

Let the initial balance in all the three accounts of the two transactions of Figure 9.9 be \$100.

An illegal schedule	Another illegal schedule
a_1: $x = 100$ b_1: $y = 100$ a_2: $z = 100$ b_2: $z = 100$ a_3: $x = 95$ b_3: $y = 93$ a_4: $z = 105$ b_4: $z = 107$	b_1: $y = 100$ b_2: $z = 100$ a_1: $x = 100$ a_2: $z = 100$ b_3: $y = 93$ b_4: $z = 107$ a_3: $x = 95$ a_4: $z = 105$
Final result	Final result
$x = 95$ $y = 93$ $z = 107$	$x = 95$ $y = 93$ $z = 105$

Time

Fig. 9.10 Two interleavings of the operations of the two transactions of Figure 9.9 that produce unexpected final results.

Let the initial balance in all the three accounts of the two transactions of Figure 9.9 be $100.

A legal schedule	Another legal schedule
a_1: $x = 100$ b_1: $y = 100$ a_2: $z = 100$ a_3: $x = 95$ a_4: $z = 105$ b_2: $y = 105$ b_3: $z = 93$ b_4: $z = 112$	b_1: $y = 100$ b_2: $z = 100$ a_1: $x = 100$ b_3: $y = 93$ b_4: $z = 107$ a_2: $z = 107$ a_3: $x = 95$ a_4: $z = 112$
Final result	Final result
$x = 95$ $y = 93$ $z = 112$	$x = 95$ $y = 93$ $z = 112$

Time

Fig. 9.11 Two interleavings of the operations of the two transactions of Figure 9.9 that produce correct final results.

- Any interleaving of the operations of two or more concurrent transactions is known as a *schedule*.
- All *schedules* that produce the same final result as if the transactions had been performed one at a time in some serial order are said to be *serially equivalent*.

Operations for Transaction-Based File Service

- In a file system that provides a transaction facility, a transaction consists of a sequence of elementary file access operations such as *read* and *write*.
- The actual operations and their sequence that constitute a transaction is application dependent, and so it is the client's responsibility to construct transactions.
- Therefore, the client interface to such a file system must include special operations for transaction service.

The three essential operations for transaction service are as follows:

begin_transaction returns (TID): Begins a new transaction and returns a unique transaction identifier (TID). This identifier is used in other operations of this transaction.
All operations within a begin transaction and an *end_transaction* form the body of the transaction.

end_transaction (TID) returns (status): This operation indicates that, from the viewpoint of the client, the transaction completed successfully.
Therefore the transaction is terminated and an attempt is made to commit it.

abort_transaction (TID): Aborts the transaction, restores any changes made so far within the transaction to the original values, and changes its status to inactive.
A transaction is normally aborted in the event of some system failure.

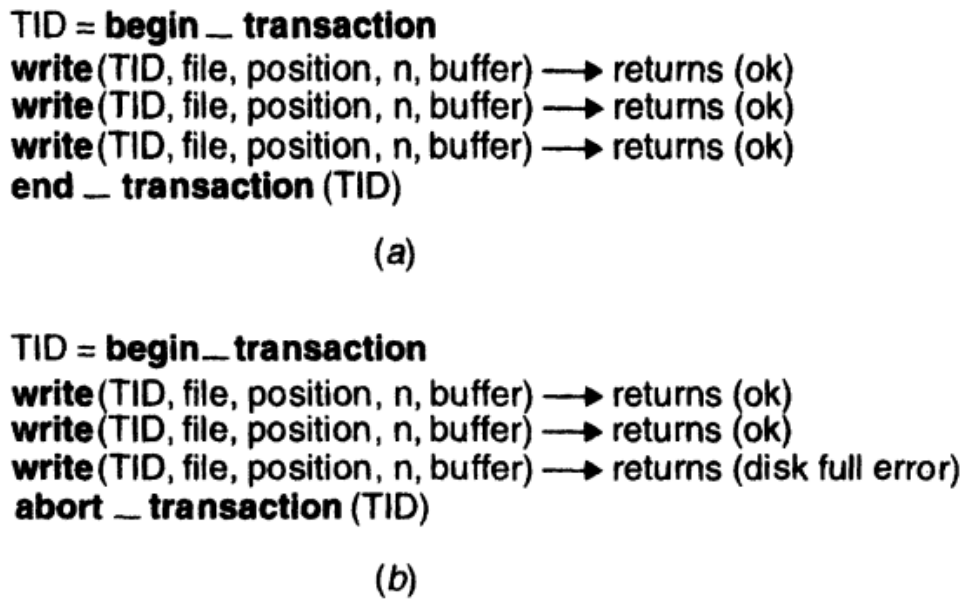

Fig. 9.12 Illustrating the use of **abort_transaction** operation: (*a*) all three operations are performed successfully so the transaction commits; (*b*) all three operations could not be performed successfully so the transaction was aborted.

- Consider the transaction of Figure 9.12, which consists of three write operations on a file.
- In Figure 9.12(a), all three operations are performed successfully, and after the end_ transaction operation, the transaction makes the new file data visible to other transactions.
- However, in Figure 9.12(b), the third write operation could not succeed because of lack of sufficient disk space.
- Therefore, in this situation, the client may use the abort_transaction operation to abort the transaction so that the results of the two write operations are undone and the file contents is restored back to the value it had before the transaction started.
- Notice that, once a transaction has been committed or aborted in the server, its state cannot be reversed by the client or the server.
- Consequently, an abort transaction request would fail if a client issued it after that transaction had been committed.

The following are file access operations for transaction service of the stateless server for the byte-stream files.

Tread (*TID*, *filename*, *position*, *n*, *buffer*): Returns to the client *n* bytes of the tentative data resulting from the *TID* if any has been recorded; otherwise it has the same effect as **Read** (*filename*, *position*, *n*, *buffer*).

Twrite (*TID*, *filename*, *position*, *n*, *buffer*): Has the same effect as **Write** (*filename*, *position*, *n*, *buffer*) but records the new data in a tentative form that is made permanent only when the *TID* commits.

Recovery Techniques

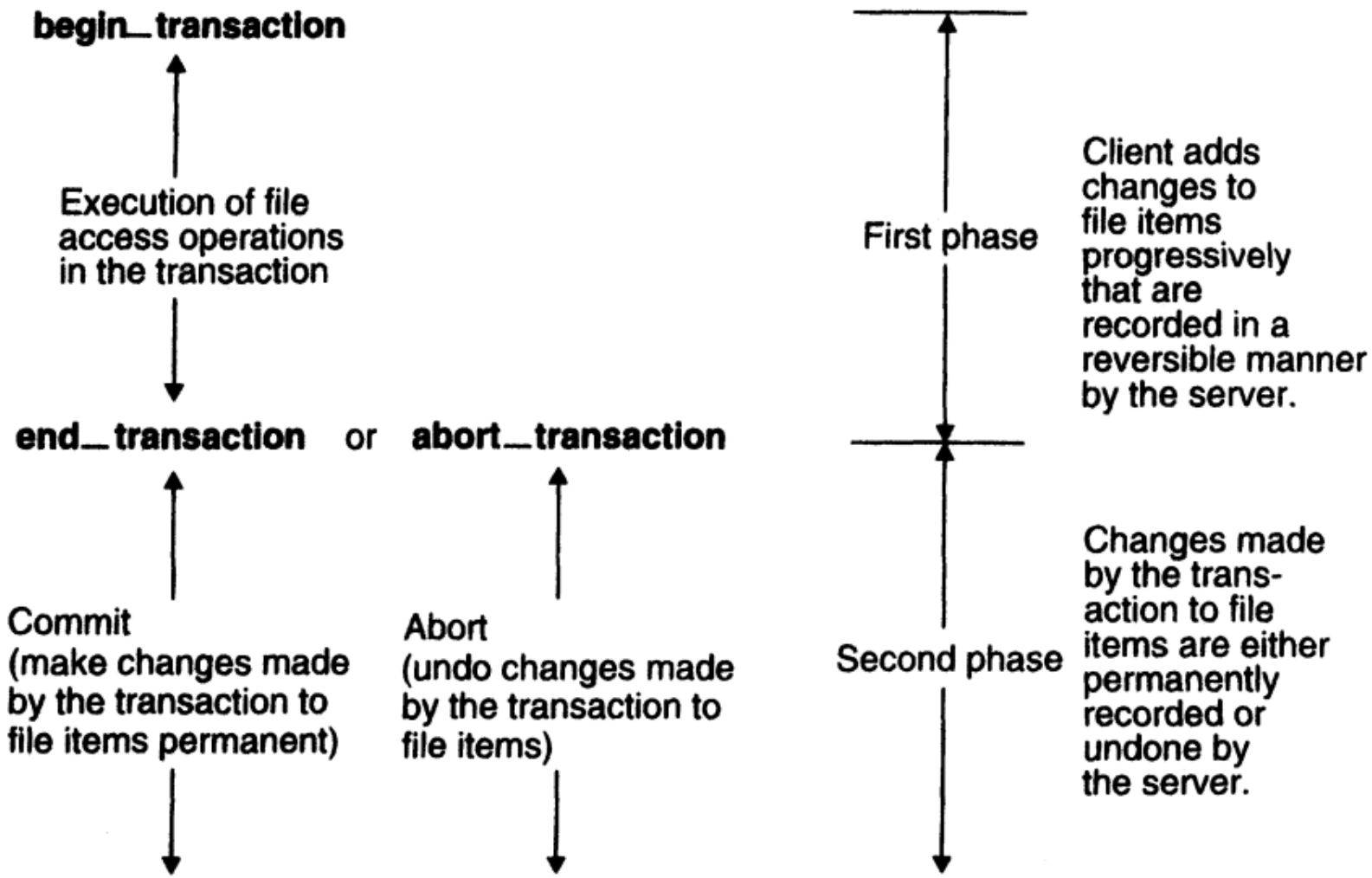

Fig. 9.13 The two phases of a transaction.

- From the point of view of a server, a **transaction has two phases** (Fig. 9.13).
- The first phase starts when the server receives a begin_transaction request from a client.

  •In this phase, the file access operations in the transaction are performed and the client adds changes to file items progressively.

  •On execution of the *end_transaction* or *abort_transaction* operation, the first phase ends and the second phase starts.

- In the second phase, the transaction is either committed or aborted.

  •In a commit, the changes made by the transaction to file items are made permanent so as to make them visible to other transactions as well.

  •On the other hand, in an abort, the changes made by the transaction to file items are undone to restore the files to the state they were in before the transaction started.

- The two commonly used approaches for recording file updates in a reversible manner are the *file versions approach* and the *write-ahead log approach*.

File Versions Approach

- A basic technique to ensure file recoverability is to avoid overwriting the actual data in physical storage. The file versions approach is based on this technique.
- When a transaction begins, the current file version is used for all file access operations (within the transaction) that do not modify the file.
- Recall that as soon as a transaction commits, the changes made by it to a file become public.
- Therefore, the current version of a file is the version produced by the most recently committed transaction.
- When the first operation that modifies the file is encountered within the transaction, the server creates a tentative version of the file for the transaction from the current file version and performs the update operation on this version of the file.
- From now on, all subsequent file access operations (read or write) within the transaction are performed on this tentative file version.

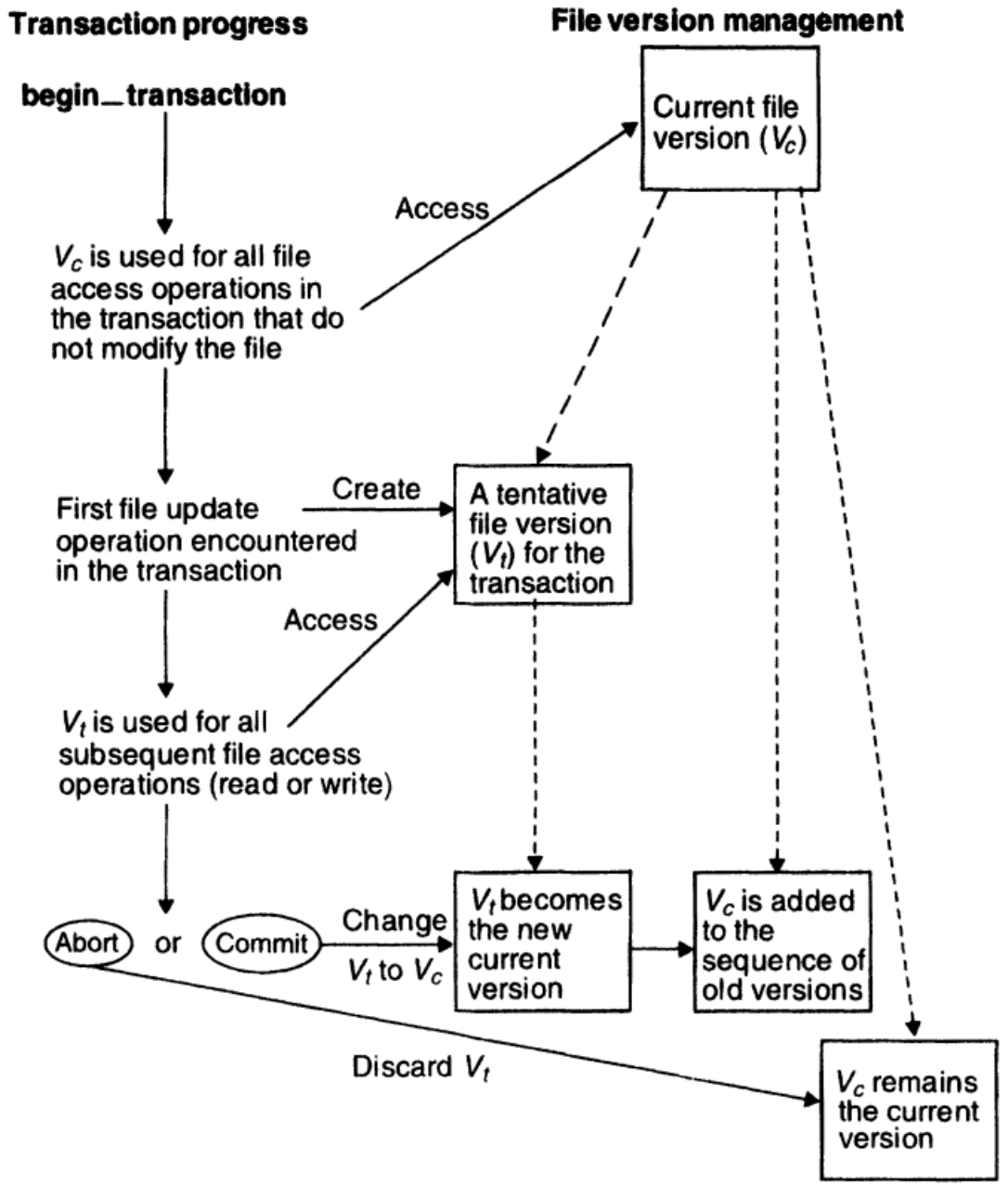

Fig. 9.14 The file versions approach for recording file updates in a reversible manner.

- When the transaction is committed, the tentative file version is made the new current version and the previous current version of the file is added to the sequence of old versions.
- On the other hand, if the transaction is aborted, the tentative file version is simply discarded and the current file version continues to remain the current version.
- A transaction can modify more than one file. In this situation, there is a tentative version of each file for the transaction.
- Since a file may be involved in several concurrent transactions, it may have several tentative versions existing at the same time.
- When one of the concurrent transactions commits, the tentative version corresponding to that transaction becomes the current version of the file.
- Since the remaining tentative versions of the file are no longer based on the current version, they are handled as follows.
  - When the next transaction out of the concurrent transactions commits, and if there are no serializability conflicts between this transaction and the previously committed transactions, the tentative version corresponding to this transaction is merged with the current version, creating a new current version that includes the changes made by all of the transactions that have. already committed.
  - If there are serializability conflicts, all the transactions that are involved except the first one to commit are aborted.

Shadow Blocks Technique for Implementing File Versions

- The tentative versions of files behave like copies of the file version from which they are derived.
- Copying the entire file for each transaction that modifies it may be wasteful and prohibitive, especially for large files.
- The shadow blocks technique is an optimization that allows the creation of a tentative version of a file without the need to copy the full file.
- It removes most of the copying.
- A file system uses some form of indexing mechanism to allocate disk space to files.
- As shown in Figure 9.15(a), in this method, the entire disk space is partitioned into fixed-length byte sequences called blocks.
- In the shadow blocks technique, a tentative version of a file is created simply by copying the index of the current version of that file.

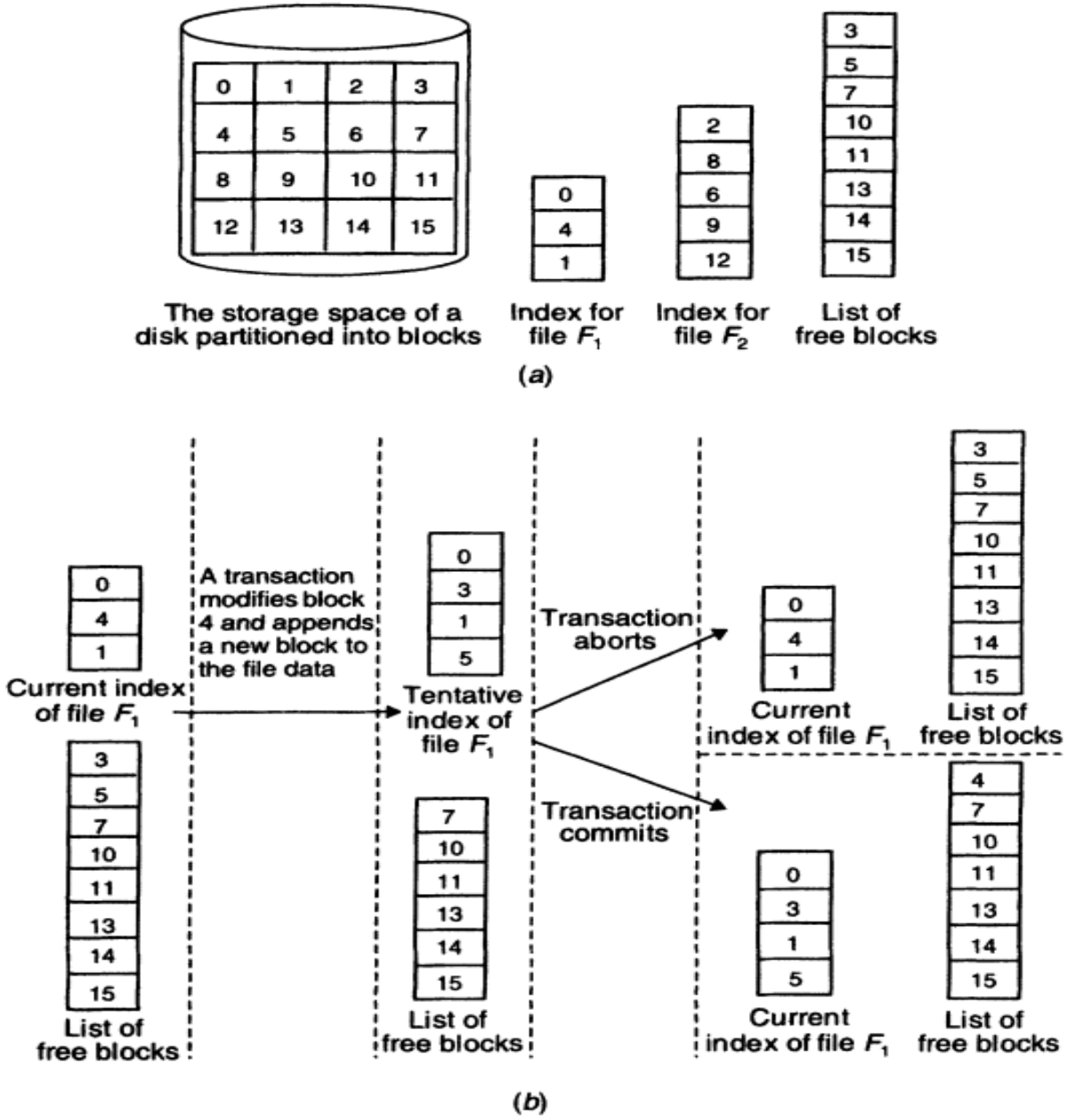

Fig. 9.15 The shadow blocks technique for implementing file versions: (*a*) example of disk blocks, file indices, and list of free blocks; (*b*) allocation and deallocation of blocks as the transaction progresses.

- The new blocks allocated to a tentative version of a file are called shadow blocks.
- Subsequent writes to the same file block by the transaction are performed on the same shadow block.
- All file access operations in the transaction are performed by using the tentative index while file access operations of other processes are performed by using the current index.
- Therefore, the process running the transaction sees the modified file version, but all other processes continue to see the original file version.
- Notice that unmodified file blocks are shared between these two file versions.
- If the transaction aborts, the shadow blocks of the tentative version of the file are returned to the list of free blocks and the tentative index is simply discarded.
- On the other hand, if the transaction commits, the tentative index is made the current index of the file, and if there is no need to retain the old file version, the blocks of the original file version whose data were modified by the transaction are added to the list of free blocks and the old current index is discarded.

The Write-Ahead Log Approach

- In this method, for each operation of a transaction that modifies a file, a record is first created and written to a log file known as a write-ahead log.
- After this, the operation is performed on the file to modify its contents.
- A write-ahead log is maintained on stable storage and contains a record for each operation that makes changes to files.
- Each record contains the identifier of the transaction that is making the modification, the identifier of the file that is being modified, the items of the file that are being modified, and the old and new values of each item modified.

- To illustrate how the log works, let us again consider the transaction for transferring $5 from account X to account Z.
- As shown in Figure 9.16, for each operation of the transaction that makes changes to file items, a record is first created and written to the write-ahead log.
- In each record, the old and new values of each item modified by the operation are separated by a slash.
- There is no need to change the file items when the transaction commits.

```
x = 100 ;
z = 100 ;
begin_transaction
    read balance (x) of account X;
    read balance (z) of account Z;
    write (x – 5) to account X;
    write (z + 5) to account Z;
end_transaction
```

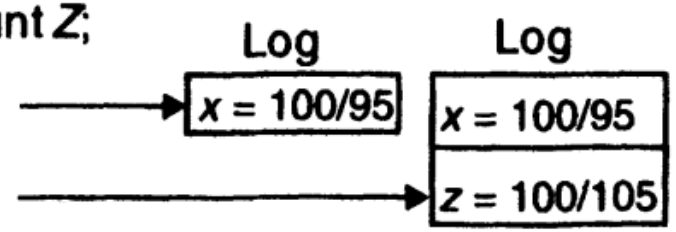

Fig. 9.16 An example of creation of write-ahead log records.

- On the other hand, if the transaction aborts, the information in the write-ahead log is used to roll back the individual file items to their initial values, thus bringing the files affected by the transaction to their original state.
- For rollback, the write-ahead log records are used one by one, starting from the last record and going backward, to undo the changes described in them.

- The write-ahead log also facilitates recovery from crashes.
- For instance, suppose the server process handling the transaction of Figure 9.16 crashes after writing the log record for the second write operation in the transaction.
- After the server's machine is restarted, the write-ahead log is used to identify the exact point of failure.
- For this, the value of the changed data item in the last record of the log (in this case z) is compared with its value in the file.
- One of the following actions are taken depending on the status of the file:
  - 1. If the value in the file is 100, it means that the crash occurred before the file was updated. Therefore, the value of the data item in the file is changed to 105.
  - 2. On the other hand, if the value in the file is 105, it means that the crash occurred after the file was updated, so nothing needs to be done in this case.

Concurrency Control

- To prevent data inconsistency due to concurrent access by multiple transactions, every transaction mechanism needs to implement a concurrency control algorithm.

- A good concurrency control mechanism allows maximum concurrency with minimum overhead while ensuring that transactions are run in a manner so that their effects on shared data are serially equivalent.

- ***The simplest approach*** for concurrency control would be to *allow the transactions to be run one at a time* so that two transactions never run concurrently and hence there is no conflict.
  - However, this approach is *not good, because it does not allow any concurrency*.

- ***Total elimination of concurrency*** *is neither acceptable nor necessary* because a transaction normally accesses only a few of the large number of files available in a system.

- ***Another simple approach to concurrency control*** - *Two transactions should be allowed to run concurrently only if they do not use a common file* (or data item in general).

- This approach is also *unnecessarily restrictive because a pair of transactions may access the same data item* in a manner that does not cause any conflict (inconsistency of the data item).
  - Also, *it is not possible to predict which data items will be used by a transaction*.

- Therefore, *more flexible concurrency control algorithms are normally used* by a transaction mechanism.

- *The most commonly used ones are* ***locking***, ***optimistic concurrency control***, and ***timestamps***.

1. Locking

- This is s *basic*, the *oldest* and the most *widely used* approach.
- Transaction locks a data item before accessing it.
- Each lock is labelled with the transaction identifier and only the transaction that locked the data item can access it any number of times.
- Other transactions that want to access the same data item must wait until the data item is unlocked.
- All data items locked by a transaction are unlocked as soon as the transaction completes (commits or aborts).
- Locking is performed by the transaction service as a part of the data access operations, and clients have no access to operations for locking or unlocking data items.

- ***Optimized Locking for Better Concurrency -*** The basic locking scheme is too restrictive and some optimizations have been proposed for better concurrency. Two of them are described below:

1. Type-specific locking.
2. Intention-to-write locks.

1. Type-specific locking.

- Used for all types of accesses to data items reduces concurrency more than is necessary.
- Better concurrency can be achieved by using type-specific locking scheme in which more than one type of locks are used based on the semantics of access operations.
- *Example:* Consider the simple case of two types of access operations *read* and *write*.
- Notice that multiple transactions that read the same data item but never write on it do not conflict.
- Therefore, when a transaction is accessing a data item in the read-only mode, there is no reason to keep those transactions waiting that also want to access the data item in the read-only mode.
- Therefore, instead of using a single lock for both read and write accesses, separate locks (read locks and write locks) should be used for the two operations.
- With these two types of locks, the locking rules are given in *Figure 9.17*.

✓ *If a read lock is set on a data item*, other read locks are permitted but write locks are not permitted.

✓Therefore, *an item locked with a read lock cannot have its value changed by other transactions*.

✓On the other hand, *when a write lock is set, no other locks of any kind are permitted.*

✓Therefore, *a data item locked with a write lock cannot be accessed* (read or written) by another transaction.

Type of lock already set	Type of lock to be set	
	Read	Write
None	Permitted	Permitted
Read	Permitted	Not permitted
Write	Not permitted	Not permitted

Fig. 9.17 Locking rules in case of read locks and write locks.

2. Intention-to-write locks

- *A transaction has two phases* and the updates made by a transaction to a data item are tentative in the first phase and are made permanent only in the second phase.

- Each transaction is unable to observe the other transactions' tentative values.

- Therefore, *when a read lock is set*, instead of preventing any other transaction from writing the locked data item, a transaction should be allowed to proceed with its tentative writes until it is ready to commit.

- The value of the item will not actually change until the writing transaction commits, so, if it is suspended at that point, the item remains unchanged until the reading transaction releases its lock.

- Based on the observation above, for better concurrency, Gifford [1979b] proposed the use of an *"intention-to-write lock" (I-write)* and a *commit lock instead of a write lock*.

- Therefore, *three types –of locks are used in this scheme*, and the locking rules for them are given in Figure 9.18.

- *If a read lock is set*, an I-write lock is permitted on the data item and vice versa.

- This is because the effects of write are not observable by any other transaction until the writing transaction commits.

- *If an I-write lock is set*, no other transaction is allowed to have an I-write lock on the same data item.

- A commit lock is not permitted if any other type of lock is already set on the data item.

Type of lock already set	Type of lock to be set		
	Read	I-write	Commit
None	Permitted	Permitted	Permitted
Read	Permitted	Permitted	Not permitted
I-write	Permitted	Not permitted	Not permitted
Commit	Not permitted	Not permitted	Not permitted

Fig. 9.18 Locking rules in case of read locks, I-write locks, and commit locks.

Two-Phase Locking Protocol

- Lock a data item for use by a transaction only for the period during which the transaction actually works on it and to release the lock as soon as the access operation is over.
- However, locking and unlocking data items precisely at the moment they are needed (or no longer needed) can lead to inconsistency problems.
- For instance, two commonly encountered problems due to early release (releasing immediately after the access operation finishes) of read locks and write locks are as follows:

1. ***Possibility of reading inconsistent data in case of two or more read accesses by the same transaction.***
   - This is because when read locks and write locks are released early, *between two subsequent read accesses to the same data item by a transaction, another transaction may update the same data item and commit*.
   - Therefore, the second read may not see the same value of the data item as the first read.
   - Notice that *if a transaction reads a data item only once per transaction, there is no harm in releasing its read lock early*.

2. Need for cascaded aborts.

- Suppose a transaction releases a write lock early and then some other transaction locks the same data item, performs all its work, and commits before the first transaction.
- Afterward, suppose the first transaction aborts.
  - In this situation, the already committed transaction must now be undone
  - Because its results are based on a value of the data item that it should never have seen.
- *Aborting of already committed transactions when a transaction aborts is known as cascaded aborting*.
- Because of the large overhead involved in cascaded aborting, it is better to avoid it.

➢ *To avoid the data inconsistency problems, transaction systems use the two-phase locking protocol.*

➢In the first phase of a transaction, known as the ***growing phase***, all locks needed by the transaction are gradually acquired.

➢Then in the second phase of the transaction, known as the ***shrinking phase***, the acquired locks are released.

➢Therefore, once a transaction has released any of its locks, it cannot request any more locks on the same or other data items.

Granularity of Locking

- The granularity of locking refers to the unit of lockable data items.
- In a file system supporting transactions, this unit is normally an entire ***file, a page, or a record***.
- If many transactions share files, the granularity of locking can have a significant impact on how many transactions can be executed concurrently.
- For instance, *if locks can be applied only to whole files, concurrency gets severely restricted due to the increased possibility of false sharing*.
- False sharing occurs when two different transactions access two unrelated data items that reside in the same file (Fig. 9.19).
- In such a situation, even though the two data items can be accessed concurrently by the two transactions, this is not allowed because the granularity of locking is a file.
- Notice that the locking granularity increases concurrency by reducing the possibility of false sharing.
- Finer locking granularity leads to
  - Larger lock management overhead,
  - Complicates implementation of recoverable files, and
  - Is more likely to lead to deadlocks.

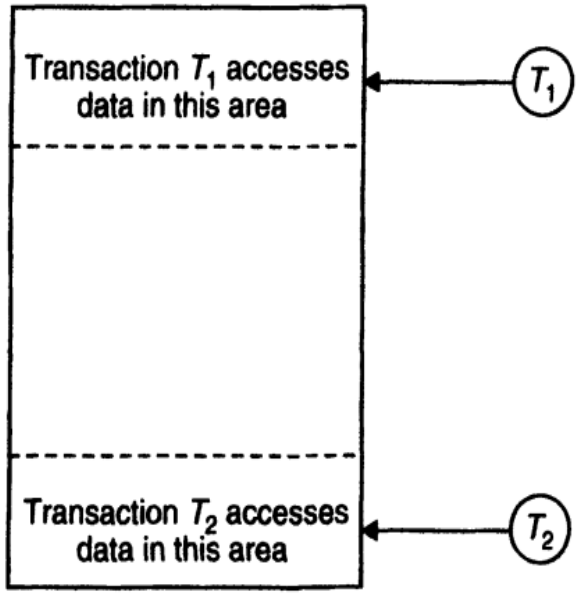

Fig. 9.19 False sharing.

Handling of Locking Deadlocks

- The locking scheme can lead to deadlocks.
- A deadlock is a state in which a transaction waits for a data item locked by another transaction that in turn waits, perhaps via a chain of other waiting transactions, for the first transaction to release some of its locks.
- Since a transaction cannot release any lock until it finishes, none of the transactions involved in such a circular wait can proceed unless one of them is aborted.

Example

- Suppose two transactions T1 and T2 have locked data items D1 and D2 , respectively.
- Now suppose that T1 requests a lock on D2 and T2 requests a lock on D1 .
- A deadlock results because each transaction has an item of data locked that the other needs to access.
- One of the following techniques may be used:

1. Avoidance
2. Detection
3. Timeouts

2. Optimistic Concurrency Control

- It is based on the observation that access conflicts in concurrently executing transactions are very rare.
- Therefore, in this approach, transactions are allowed to proceed uncontrolled up to the end of the first phase.
- However, in the second phase, before a transaction is committed, the transaction is validated to see if any of its data items have been changed by any other transaction since it started.
- The transaction is committed if found valid; otherwise it is aborted.
- For the validation process, two records are kept of the data items accessed within a transaction-a ***read set*** that contains the data items read by the transaction and a ***write set*** that contains the data items changed, created, or deleted by the transaction.
- To validate a transaction, its read set and write set are compared with the write sets of all of the concurrent transactions that reached the end of their first phase before it.
- The validation fails if any data item present in the read set or write set of the transaction being validated is also present in the write set of any of the concurrent transactions mentioned above.

Two main advantages of the optimistic concurrency control approach are as follows:

1. It allows maximum parallelism because all transactions are allowed to proceed independently in parallel without any need to wait for a lock.
2. It is free from deadlock.

However, it suffers from the following drawbacks:

1. It requires that old versions of files corresponding to recently committed transactions be retained for the validation process. This is not necessary either with locking or timestamping.
2. Although the approach is free from deadlock, it may cause the starvation of a transaction. This is because a transaction that fails validation is aborted and then restarted all over again. But if the transaction comes into conflict with other transactions for the use of data items each time it is restarted, it can never pass the validation checks.
   - ✓ To solve the starvation problem, Kung and Robinson suggested that the server should detect a transaction that has been aborted several times.
   - ✓ When such a transaction is detected, it should be given exclusive access to the data items it uses by the use of a critical section protected by a semaphore
3. In an overloaded system, the number of transactions getting aborted due to access conflicts may go up substantially, resulting in increased overhead for rerunning the aborted transactions. Optimistic concurrency control is not a suitable approach for such situations.
   1. Mullender and Tanenbaum [1985] suggested that Locking should be used in transactions in which several files are changed and where the chance of two transactions using the same data item is high.
   2. On the other hand, optimistic concurrency control should be used for transactions using one file and in which the likelihood of two transactions accessing the same data item is low.

❖ In the optimistic concurrency control approach, a transaction is validated only after it has completed its first phase.

❖ If the validation fails because some conflict is detected, the transaction is aborted and restarted all over again.

❖ Execution of operations of the transaction that follow the operation that caused the conflict is actually a waste in this case.

❖ The overhead involved in executing the operations of the transaction that follow the operation that caused the conflict can be totally avoided if it is possible to detect the conflict right at the time when the operation causing it is executed.

❖ This is because the transaction can be aborted immediately at the point where the conflict occurs and it can then be restarted.

❖This has been made possible in the **timestamps** approach, in which each operation in a transaction is validated when it is carried out.

❖ If the validation fails, the transaction is aborted immediately and it can then be restarted.

3. Timestamps

- To perform validation at the operation level, each transaction is assigned a unique timestamp at the moment it does begin_transaction.
- In addition, every data item has a read timestamp and a write timestamp associated with it.
- When a transaction accesses a data item, depending on the type of access (read or write), the data item's read timestamp or write timestamp is updated to the transaction's timestamp.
- As usual, the write operations of a transaction are recorded tentatively and are invisible to other transactions until the transaction commits.
- Therefore, when a transaction is in progress, there will be a number of data items with tentative values and write timestamps.
- The tentative values and timestamps become permanent when the transaction commits.
- Before performing a read operation or a write operation on a data item, the server performs a validation check by inspecting the timestamps on the data item, including the timestamps on its tentative values that belong to incomplete transactions.
- The rules for validation are as follows:

Validation of a Write Operation:-

If the timestamp of the current transaction (transaction that requested the write operation) is either equal to or more recent than the read and (committed) write timestamps of the accessed data item, the write operation passes the validation check.

- Therefore a tentative write operation is performed in this case.
- On the other hand, if the timestamp of the current transaction is older than the timestamp of the last read or committed write of the data item, the validation fails.
- This is because another transaction has accessed the data item since the current transaction started.
- Therefore the current transaction is aborted in this case.

NOTE:

i. A transaction can complete its first phase only if all its operations have been consistent with those of earlier transactions.

ii. Therefore, if a transaction completes its first phase, it can always be committed, although it may have to wait for earlier transactions that have tentative copies of shared data items to commit.

iii. The timestamp-based concurrency control scheme described above is used in the SDD-I database system [

iv. Timestamp-based concurrency control schemes are deadlock free.

Validation of a Read Operation.

- If the timestamp of the current transaction (transaction that requested the read operation) is more recent than the write timestamps of all committed and tentative values of the accessed data item, the read operation passes the validation check.

- However, the read operation can be performed immediately only if there are no tentative values of the data item; otherwise it must wait until the completion of the transactions having tentative values of the data item.

- On the other hand, the validation check fails and the current transaction is aborted in the following cases:

    1. The timestamp of the current transaction is older than the timestamp of the most recent (committed) write to the data item.

    2. The timestamp of the current transaction is older than that of a tentative value of the data item made by another transaction, although it is more recent than the timestamp of the permanent data item.