UNIT 4

Lecture 38

Transaction Processing

Dinesh Kumar Bhawnani

Bhilai Institute of Technology, DURG

Transaction

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- Usually, a transaction is initiated by a user program written in a high-level data manipulation language or programming language (for example, SQL, COBOL, C, C++, or Java), where it is delimited by statements (or function calls) of the form begin transaction and end transaction.
- The transaction consists of all operations executed between the begin transaction and end transaction.

ACID Properties of Transaction

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

Atomicity – Either all operations of the transaction are reflected property in the database, or none are.

Consistency – Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

Isolation – Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

Durability – After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

ACID Properties of Transaction

- Transaction access data using two operations:
    1. read(X), which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
    2. write(X), which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

- For e.g., consider a simplified banking system consisting of several accounts and a set of transactions that access and update those accounts. Let T_i be a transaction that transfers $50 from account A to account B. This transaction can be defined as

```
Ti:     read(A);
        A:=A-50;
        write(A);
        read(B);
        B:=B+50;
        write(B).
```

Consistency

- The consistency requirement here is that is that the sum of A and B be unchanged by the execution of the transaction.
- Without the consistency requirement, money could be created or destroyed by the transaction. It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.
- Ensuring consistency for an individual transaction is the responsibility of the **application programmer** who codes the transaction.

Atomicity

- Suppose that, just before the execution of transaction T_i the values of accounts A and B are \$1000 and \$2000, respectively. Now suppose that, during the execution of transaction T_i, a failure occurs that prevents T_i from completing its execution successfully. These failures may be power failures, hardware failures or software errors. Further, suppose that the failure happened after the write(A) operation but before the write(B) operation. In this case, the values of accounts A and B reflected in the database are \$950 and \$2000. The system destroyed \$50 as result of this failure.
- The basic idea behind ensuring atomicity is that the database system keeps track of the old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed.
- Ensuring atomicity is the responsibility of a component of the database system called the **transaction – management component**.

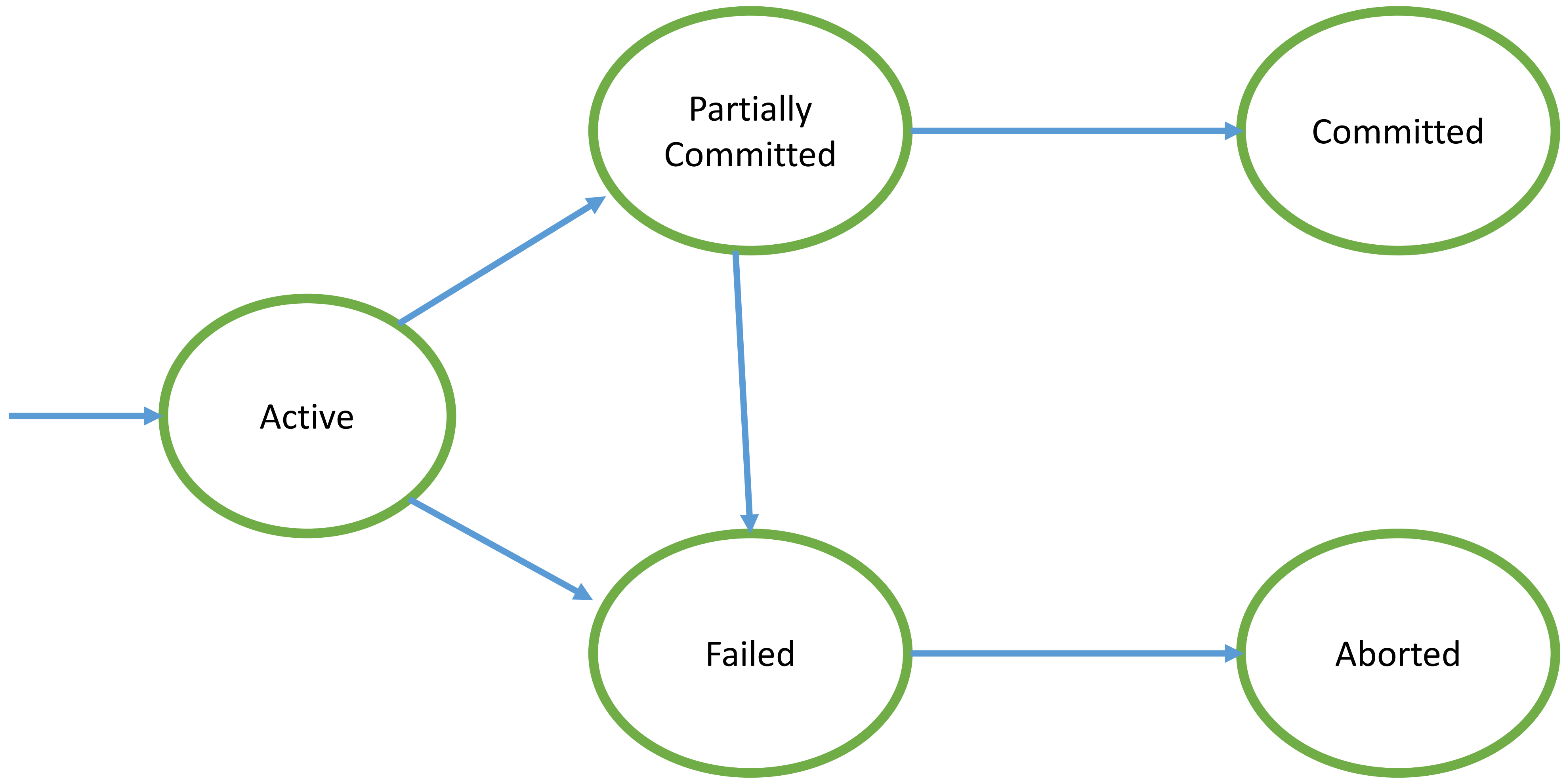
Durability

- Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds.
- The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.
- We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either
    - The updates carried out by the transaction have been written to disk before the transaction completes.
    - Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.
- Ensuring durability is the responsibility of a component of the database system called the **recovery – management component**.

Isolation

- Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.
- For e.g., In T_i the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value.
- A way to avoid the problem of concurrently executing transactions is to execute transactions serially, i.e. one after the other.
- The isolation property of a transaction ensures that the concurrently execution of the transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order.
- Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency – control component**.

State Diagram of a transaction



Transaction States

- Sometimes it so happens that a transaction is not completed successfully.
- Such a transaction is called **aborted**.
- To ensure the atomicity property, any changes that the aborted transaction has made to the database are undone. When the changes made by an aborted transaction to the database are undone, the transaction is said to have been **rolled back**.
- It is part of the responsibility of the **recovery scheme** to manage transaction aborts.
- The transaction that has completed its execution successfully is said to be **commited**.
- If a transaction has committed, its effect cannot be undone.

Transaction States

A transaction must be in one of the following states:

1. **Active** – It is the initial state. While the transaction is being executed, it stays in active state.
2. **Partially committed** – Transaction becomes partially committed after its final statement has been executed.
3. **Failed** – When the normal execution can no longer be carried on.
4. **Aborted** – It is the state after the transaction has been rolled back and database has been restored to the state prior to the transaction’s starting.
5. **Committed** – State after the successful completion of transaction.

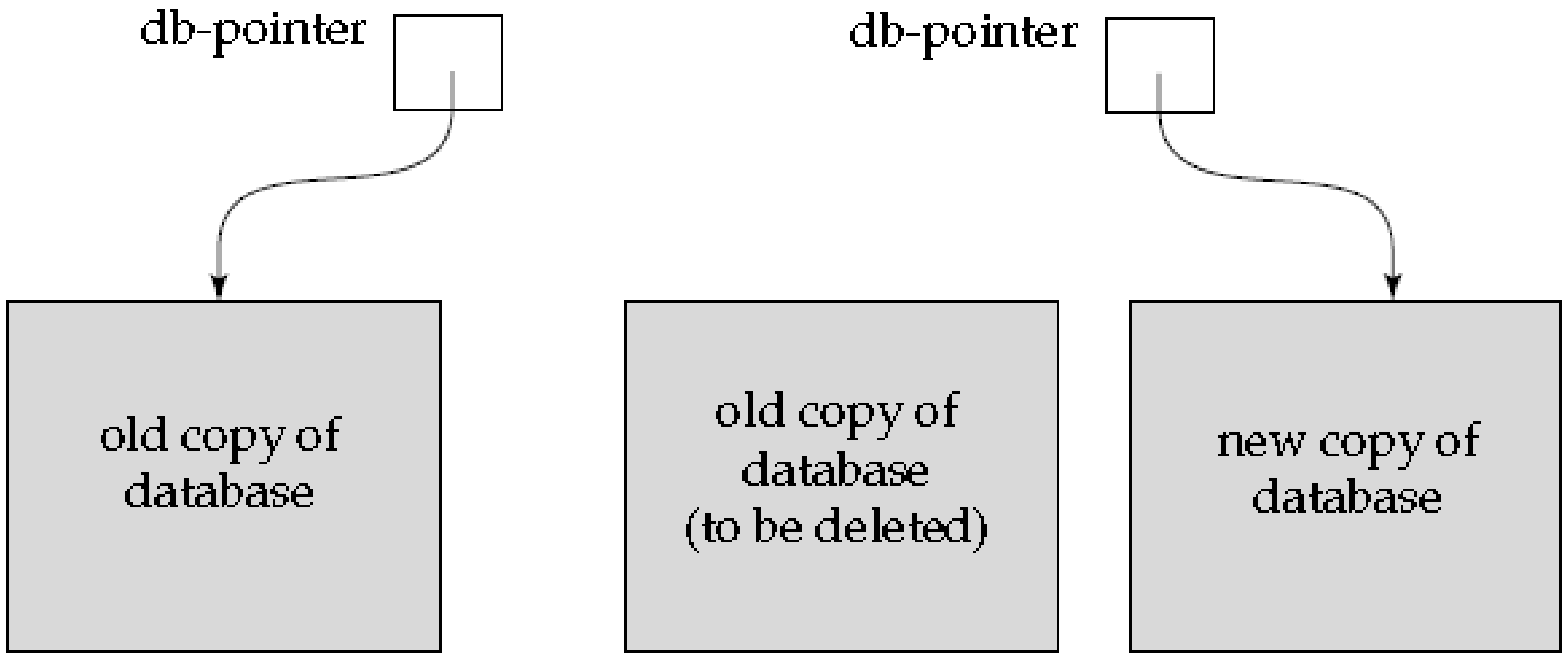
Transaction Processing in various states

- We say that a transaction has committed only if it has entered the committing state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if it has either committed or aborted.
- A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.
- The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be recreated when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

Transaction Processing in various states

- A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution due to some hardware or logical errors. Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options :
    1. It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software errors that were not created though the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
    2. It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

Shadow Copy scheme to ensure Atomicity and Durability



Shadow Copy scheme to ensure Atomicity and Durability

- The recovery-management component of a database system can support atomicity and durability by a variety of schemes.
- This scheme, which is based on making copies of the database, called *shadow* copies, assumes that only one transaction is active at a time.
- The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.
- In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the **shadow copy**, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.

Shadow Copy scheme to ensure Atomicity and Durability

- If the transaction completes, it is committed as follows.
  1. First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk.
  2. After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database.
  3. The old copy of the database is then deleted.
- The transaction is said to have been *committed* at the point where the updated db-pointer is written to disk.

Shadow Copy scheme to ensure Atomicity and Durability

- We now consider how the technique handles transaction and system failures.
  1. First, consider transaction failure.
  2. If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected. We can abort the transaction by just deleting the new copy of the database.
  3. Once the transaction has been committed, all the updates that it performed are in the database pointed to by db-pointer. Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

For Video lecture on this topic please subscribe to my youtube channel.

The link for my youtube channel is

https://www.youtube.com/channel/UCRWGtE76JlTp1iim6aOTRuw?sub_confirmation=1