Chapter 7
Transaction Management

ACID Properties

ACID (atomicity, consistency, isolation, durability) is a set of properties that guarantee database transactions are processed reliably. The concept of ACID is to evaluate databases and application architecture. In the context of databases, a single logical operation on the data is called a transaction.

Atomicity

Atomicity requires that database modifications must follow an "all or nothing" rule. Each transaction is said to be atomic if when one part of the transaction fails, the entire transaction fails and database state is left unchanged. It is critical that the database management system maintains the atomic nature of transactions in spite of any application, DBMS, operating system or hardware failure.

An atomic transaction cannot be subdivided, and must be processed in its entirety or not at all. Atomicity means that users do not have to worry about the effect of incomplete transactions.

Transactions can fail for several kinds of reasons:

1. Hardware failure: A disk drive fails, preventing some of the transactions on database changes from taking effect
2. System failure: The users lose their connection to the application before providing all necessary information
3. Database failure: E.g., the database runs out of room to hold additional data
4. Application failure: The application attempts to post data that violates a rule that the database itself enforces, such as attempting to create a new account without supplying an account number.

Atomicity failure

The transaction subtracts 10 from A and adds 10 to B. If it succeeds, it would be valid because the data continues to satisfy the constraint. However, assume that after removing 10 from A, the transaction is unable to modify B. If the database retains A's new value, atomicity and the constraint would both be violated. Atomicity requires that both parts of this transaction complete or neither.

Consistency

The consistency property ensures that the database remains in a consistent state; more precisely, it says that any transaction will take the database from one consistent state to another consistent state.

The consistency property does not say how the DBMS should handle an inconsistency other than ensure the database is clean at the end of the transaction. If, for some reason, a transaction is executed that violates the database’s consistency rules, the entire transaction could be rolled back to the pre-transactional state - or it would be equally valid for the DBMS to take some patch-up action to get the database in a consistent state. Thus, if the database schema says that a particular field is for holding integer numbers, the DBMS could decide to reject attempts to put fractional values there, or it could round the supplied values to the nearest whole number: both options maintain consistency.

The consistency rule applies only to integrity rules that are within its scope. Thus, if a DBMS allows fields of a record to act as references to another record, then consistency implies the DBMS must enforce referential integrity: by the time any transaction ends, each and every reference in the database must be valid. If a transaction consisted of an attempt to delete a record referenced by another, each of the following mechanisms would maintain consistency:

- abort the transaction, rolling back to the consistent, prior state;
- delete all records that reference the deleted record (this is known as *cascade delete*); or,
- Nullify the relevant fields in all records that point to the deleted record.

These are examples of Propagation constraints; some database systems allow the database designer to specify which option to choose when setting up the schema for a database.

Application developers are responsible for ensuring application level consistency, over and above that offered by the DBMS. Thus, if a user withdraws funds from an account and the new balance is lower than the account's minimum balance threshold, as far as the DBMS is concerned, the database is in a consistent state even though this rule (unknown to the DBMS) has been violated.

Consistency failure

Consistency is a very general term that demands the data meets all validation rules that the overall application expects - but to satisfy the consistency property a database system only needs to enforce those rules that are within its scope. In the previous example, one rule was a requirement that A + B = 100; most database systems would not allow such a rule to be specified, and so would have no responsibility to enforce it - but they would be able to ensure the values were whole numbers. Example of rules that can be enforced by the database system are that the primary keys values of a record uniquely identify that record, that the values stored in fields are the right type (the schema might require that both A and B are integers, say) and in the right range, and that foreign keys are all valid.

Isolation

Isolation refers to the requirement that other operations cannot access data that has been modified during a transaction that has not yet completed. The question of isolation occurs in case of concurrent transactions, (i.e. multiple transactions occurring at the same time) and within the same database. Each transaction must remain unaware of other concurrently executing transactions, except that one transaction may be forced to wait for the completion of another transaction that has modified data that the waiting transaction requires. If the isolation system does not exist, then the data could be put into an inconsistent state. This could happen if one transaction is in the process of modifying data but has not yet completed, and then a second transaction reads and modifies that uncommitted data from the first transaction. If the first transaction fails and the second one succeeds, that violation of transactional isolation will cause data inconsistency. Due to performance and deadlocking concerns with multiple competing transactions, many modern databases allow dirty reads which is a way to bypass some of the restrictions of the isolation system. A dirty read means that a transaction is allowed to read - but not modify - the uncommitted data from that of another transaction.

Isolation failure

To demonstrate isolation, we assume two transactions execute at the same time, each attempting to modify the same data. One of the two must wait until the other completes in order to maintain isolation.

Consider two transactions. T1 transfers 10 from A to B. T2 transfers 10 from B to A. Combined, there are four actions:

- subtract 10 from A
- add 10 to B.
- subtract 10 from B
- add 10 to A.

If these operations are performed in order, isolation is maintained, although T2 must wait. Consider what happens if T1 fails half-way through. The database eliminates T1's effects, and T2 sees only valid data.

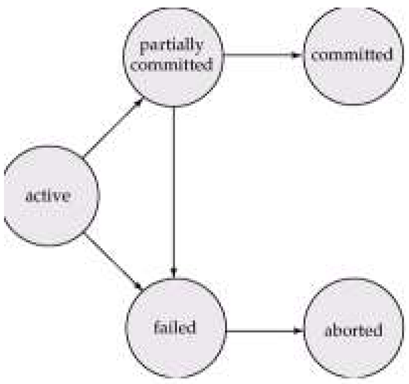
By interleaving the transactions, the actual order of actions might be: A-10, B-10, B+10, A+10. Again consider what happens if T1 fails. T1 still subtracts 10 from A. Now T2 adds 10 to A restoring it to its initial value. Now T1 fails. What should A's value be? T2 has already changed it. Also, T1 never changed B. T2 subtracts 10 from it. If T2 is allowed to complete, B's value will be 10 too low, and A's value will be unchanged, leaving an invalid database. This is known as a write-write failure because two transactions attempted to write to the same data field.

<u>Durability</u>

Durability is the ability of the DBMS to recover the committed transaction updates against any kind of system failure (hardware or software). Durability is the DBMS's guarantee that once the user has been notified of a transaction's success the transaction will not be lost, the transaction's data changes will survive system failure, and that all integrity constraints have been satisfied, so the DBMS won't need to reverse the transaction. Many DBMSs implement durability by writing transactions into a transaction log that can be reprocessed to recreate the system state right before any later failure. A transaction is deemed committed only after it is entered in the log.

Durability does not imply a permanent state of the database. A subsequent transaction may modify data changed by a prior transaction without violating the durability principle

<u>Transaction States</u>



A database transaction can be in one of the following four states:

- **Active State:** A database transaction is in this phase while its statements start to be executed. The transaction is in this state immediately when the queries are fired.
- **Partially Committed Phase:** a database transaction enters this phase when its final statement has been executed. At this phase, the database transaction has finished its execution, but it is still possible for the transaction to be aborted because the output from the execution may remain residing temporarily in main memory - an event like hardware failure may erase the output.
- **Failed State:** A database transaction enters the failed state when its normal execution can no longer proceed due to hardware or program errors).
- **Aborted State:** A database transaction, if determined by the DBMS to have failed, enters the aborted state. An aborted transaction must have no effect on the database, and thus any changes it made to the database have to be undone, or in technical terms, ***rolled back***. The database will return to its consistent state when the aborted transaction has been rolled back. The DBMS's recovery scheme is responsible to manage transaction aborts.
- **Committed State:** A database transaction enters the committed state when enough information has been written to disk after completing its execution with success. In this state, so much information has been written to disk that the effects produced by the transaction cannot be undone via aborting; even when a system failure occurs, the changes made by the committed transaction can be re-created when the system restarts.

Concurrency Control

Concurrency control in database management systems (DBMS) ensures that database transactions are performed concurrently without the concurrency violating the data integrity of the respective databases. Thus concurrency control is an essential component for correctness in any system where two database transactions or more can access the same data concurrently, e.g., virtually in any general-purpose database system. A database transaction is defined as an object that meets the ACID rules. A DBMS usually guarantees that only serial transaction schedules are generated, for correctness; i.e., transactions are executed serially, one after another, with no overlap in time. For maintaining correctness in cases of failed transactions schedules also need to be recovered. A DBMS also guarantees that no effect of committed transactions is lost, and no effect of aborted (rolled back) transactions remains in the related database.

Concurrency Methods:

Schedules

• Transactions must have scheduled so that data is serially equivalent
• Use mutual exclusion to ensure that only one transaction executes at a time
    or…
• Allow multiple transactions to execute concurrently
  – but ensure serializability
• Concurrency control

Locking
• Serialize with exclusive locks on a resource
  – lock data that is used by the transaction
  – lock manager
• Conflicting operations of two transactions must be executed in the same order
  – transaction not allowed new locks after it has released a lock
• Two-phase locking
  – phase 1: growing phase: acquire locks
  – phase 2: shrinking phase: release locks

Multiple readers/single writer
• Improve concurrency by supporting multiple readers
  – there is no problem with multiple transactions *reading* data from the same object
  – only one transaction should be able to write to an object
  – and no other transactions should read in this case
• Two locks: *read locks* and *write locks*
  – set a *read lock* before doing a read on an object
  – set a *write lock* before doing a write on an object
  – block (wait) if transaction cannot get the lock
• If a transaction has
– no locks for an object: another transaction may obtain a *read* or *write* lock
– a read lock for an object: another transaction may obtain a *read* but wait for *write* lock
– a write lock for an object: another transaction will have to wait for a *read* or a *write* lock

Timestamp ordering
• Assign unique timestamp to a transaction when it begins
• Each object has a *read* and *write* timestamp associated with it
  – which *committed* transaction last read the object
  – which *committed* transaction last wrote the object
• *Good ordering*:
  – object's *read* and *write* timestamps will be older than current transaction if it wants to write an object
  – object's *write* timestamps will be older than current transaction if it wants to read an object
• Abort and restart transaction for improper ordering

Database Recovery

A computer system, like any other mechanical or electrical system is subject to failure. There are a variety of causes, including disk crash, power failure, software errors, a fire in the machine room, or even sabotage. Whatever the cause, information may be lost. The database must take actions in advance to ensure that the atomicity and durability properties of transactions are preserved. An integral part of a database system is a recovery scheme that is responsible for the restoration of the database to a consistent stage that existed prior to the occurrence of the failure.

Failure Classification

The major types of failures involving data integrity (as opposed to data security) are:

- **Transaction Failure**:
    - **Logical error**. The transaction cannot continue with its normal execution because of such things as bad input, data not found, or resource limit exceeded.
    - **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.

- **System Crash.** There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of the nonvolatile storage remains intact, and is not corrupted.

- **Disk Failure.** A disk block loses its contents as a result of either a head crash or failure during a data transfer. Copies of data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure.

Storage Structure

- **Volatile storage**. Information here does not usually survive system crashes.
- **Nonvolatile storage.** This information normally does survive system crashes, but can be lost (in a head crash, etc).
- **Stable storage.** System designed not to loss data.

1. **Loss of Volatile Storage**:
    - Can be recovered using the following methods:
        - Log-based recovery
        - Buffer management
        - Checkpoints
        - Shadow paging techniques
        -

2. **Loss of Non-Volatile Storage**:
    - Can be recovered using the following methods:
        - Periodically dump (full backup) to a stable storage
        - If a failure occurs, most recent dump is used in restoring the database.

Stable Storage Implementation

To implement stable storage, we need to replicate the needed information on several nonvolatile media with independent failure modes and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

RAID systems guarantee that the failure of a single disk will not result in the loss of data. The simplest and fasted form of RAID is the mirrored disk, which keeps two copies of each block, on separate disks. RAID systems cannot guarantee failure of a site! The most secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing it on a local disk system.

Cascading Rollback

A cascading rollback occurs in database systems when a transaction (T1) causes a failure and a rollback must be performed. Other transactions dependent on T1's actions must also be rolled back due to T1's failure, thus causing a cascading effect. That is, one transaction's failure causes many to fail.

Recoverable Schedules

A schedule (also called history) of a system is an abstract model to describe execution of transactions running in the system. Often it is a list of operations (actions) ordered by time, performed by a set of transactions that are executed together in the system. If order in time between certain operations is not determined by the system, then a partial order is used. Examples of such operations are requesting a read operation, writing, aborting, committing, requesting lock, locking, etc. Schedules and schedule properties are fundamental concepts in database concurrency control theory.

Log-Based Recovery

The most widely used structure for recording database modifications is the log. The log is a sequence of log records and maintains a history of all update activities in the database. There are several types of log records.

An update log record describes a single database write:

- Transactions identifier.
- Data-item identifier.
- Old value.
- New value.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification that has already been output to the database. Also we have the ability to undo a modification that has already been output to the database, by using the old-value field in the log records.
For log records to be useful for recovery from system and disk failures, the log must reside on stable storage. However, since the log contains a complete record of all database activity, the volume of data stored in the log may become unreasonable large.

Deferred Database Modification

The deferred-modification technique ensures transaction atomicity by recording all database modifications in the log, but deferring all write operations of a transaction until the transaction partially commits (i.e., once the final action of the transaction has been executed). Then the information in the logs is used to execute the deferred writes. If the system crashes or if the transaction aborts, then the information in the logs is ignored.

Immediate Database Modification

The immediate-update technique allows database modifications to be output to the database while the transaction is still in the active state. These modifications are called uncommitted modifications. In the event of a crash or transaction failure, the system must use the old-value field of the log records to restore the modified data items.

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. Rather than reprocessing the entire log, which is time-consuming and much of it unnecessary, we can use checkpoints:
Output onto stable storage all the log records currently residing in main memory.
Output to the disk all modified buffer blocks.
Output onto stable storage a log record, <checkpoint>.
Now recovery will be to only process log records since the last checkpoint record.

Shadow Paging

Shadow paging is an alternative to log-based recovery techniques, which has both advantages and disadvantages. It may require fewer disk accesses, but it is hard to extend paging to allow multiple concurrent transactions. The paging is very similar to paging schemes used by the operating system for memory management.
The idea is to maintain two page tables during the life of a transaction: the current page table and the shadow page table. When the transaction starts, both tables are identical. The shadow page is never changed during the life of the transaction. The current page is updated with each write operation. Each table entry points to a page on the disk. When the transaction is committed, the shadow page entry becomes a copy of the current page table entry and the disk block with the old data is released. If the shadow is stored in nonvolatile memory and a system crash occurs, then the shadow page table is copied to the current page table. This guarantees that the shadow page table will point to the database pages corresponding to the state of the database prior to any transaction that was active at the time of the crash, making aborts automatic.

There are drawbacks to the shadow-page technique:

- **Commit overhead.** The commit of a single transaction using shadow paging requires multiple blocks to be output -- the current page table, the actual data and the disk address of the current page table. Log-based schemes need to output only the log records.
- **Data fragmentation.** Shadow paging causes database pages to change locations (therefore, no longer contiguous.
- **Garbage collection.** Each time that a transaction commits, the database pages containing the old version of data changed by the transactions must become inaccessible. Such pages are considered to be *garbage* since they are not part of the free space and do not contain any usable information. Periodically it is necessary to find all of the garbage pages and add them to the list of free pages. This process is called *garbage collection* and imposes additional overhead and complexity on the system.

Recovery with Concurrent Transactions

Regardless of the number of concurrent transactions, the disk has only one single disk buffer and one single log. These are shared by all transactions. The buffer blocks are shared by transactions. We allow immediate updates, and permit a buffer block to have data items updated by one or more transactions.

Buffer Management

Log-Record Buffering

The cost of performing the output of a block to stable storage is sufficiently high that it is desirable to output multiple log records at once, using a buffer. When the buffer is full, it is output with as few output operations as possible. However, a log record may reside in only main memory for a considerable time before it is actually written to stable storage. Such log records are lost if the system crashes. It is necessary, therefore, to write all buffers related to a transaction when it is committed. There is no problem written the other uncommitted transactions at this time.

Database Buffering

Database buffering is the standard operating system concept of virtual memory. Whenever blocks of the database in memory must be replaced, all modified data blocks and log records associated with those blocks must be written to the disk.

Operating System Role in Buffer Management

We can manage the database buffer sing one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that the DBMS manages instead of the operating system. This means that the buffer must be kept as small as possible (because of its impact on other processes active on the CPU) and it adds to the complexity of the DBMS.
2. The DBMS implements its buffer within the virtual memory of the operating system. The operating system would then have to coordinate the swapping of pages to insure that the appropriate buffers were also written to disk. Unfortunately, almost all current-generation operating systems retain complete control of virtual memory. The operating system reserves space on disk for storing virtual memory pages that are not currently in main memory, called swap space. This approach may result in extra output to the disk.