Transactions

A **transaction** is a logical unit of DB processing that must be completed in its **entirety** to ensure correctness.

A transaction includes one or more DB access operations:

* Insertion
* Deletion
* Modification
* Retrieval

|  |
| --- |
| BEGIN\_TRANSACTION  READ or WRITE operation 1  READ or WRITE operation 2  …  END\_TRANSACTION  COMMIT or ROLLBACK / ABORT |

A transaction may fail because of:

* System crash
* Transaction error (div by zero, incorrect attribute reference etc.)
* Checks (insufficient bank funds)

All transactions should possess the **ACID** properties:

|  |  |
| --- | --- |
| Atomicity | A transaction should be performed in its **entirety** or not performed at all. |
| Consistency preservation | A transaction should preserve the **consistency** of the database.  It should take the database from one consistent state to another. |
| Isolation | A transaction should appear as though it is being executed in **isolation**.  The execution of a transaction should not be **interfered** with by any other transactions executing concurrently. |
| Durability | Changes applied by a committed transaction must **persist** in the database.  These changes must **not** be lost because of any failure. |

Concurrency Control

**Concurrency control** is used when multiple operations try to access the same data at the same time.

### Concurrent Transaction Issues

**Lost update** is when an incorrect value is written to the database.

**Temporary update** (dirty read) is when one transaction fails and must roll back.

**Incorrect summary** is when one transaction is calculating an aggregate summary on attributes that are currently being updated by another transaction.

Schedules

A **schedule** determines the **order of execution** of operations in concurrent transactions.

Operations from **different** transactions may be interleaved.

Sa: Oi(X); Oi(Y); Oj(X); Oj(Y);

where Oi(X) indicates a read or write executed by transaction Ti on data item X.

Schedule shorthand:

* b BEGIN\_TRANSACTION
* r Read item
* w Write item
* e END\_TRANSACTION
* c COMMIT
* a ABORT / ROLLBACK

Example:

|  |  |
| --- | --- |
| Transaction 1 | Transaction 2 |
| read(reserved\_seats\_X)  reserved\_seats\_X -= n  write(reserved\_seats\_X)  read(reserved\_seats\_Y)  reserved\_seats\_Y += n  write(reserved\_seats\_Y) | read(reserved\_seats\_X)  reserved\_seats\_X += m  write(reserved\_seats\_X) |

S1: R1(reserved\_seats\_X); W1(reserved\_seats\_X);

R2(reserved\_seats\_X); W2(reserved\_seats\_X);

R1(reserved\_seats\_Y); W1(reserved\_seats\_Y);

Two operations in a schedule **conflict** if:

* They belong to **different** transactions.
* They access the **same** data item, and at least one of the operations is a **write**.
* Changing the **order** of the operations can result in a different outcome.

Serializability

In a **serial schedule**, the operations of each transaction are executed consecutively without interleaved operations. One transaction is active at a time.

If a non-serial schedule meets the following criteria, it is said to be serializable:

1. Result equivalent
2. Conflict equivalent

Two schedules are **result equivalent** if they produce the same **final state** of the DB.

Two schedules are **conflict equivalent** if the **order** of any two conflicting operations is the same in both schedules.

Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Schedule A | | = | Schedule B | |
| read(X)  X = X - n  write(X)  read(Y)  Y = Y + n  write(Y) | read(X)  X = X + m  write(X) | read(X)  X = X - n  write(X)  read(Y)  Y = Y + n  write(Y) | read(X)  X = X + m  write(X) |

All serial schedules are **correct**.

Concurrency Control Protocols

Most DBMSs implement a set of **protocols** that must be followed for every transaction:

|  |  |
| --- | --- |
| Locking Protocols | Data items are **locked** to prevent multiple transactions from accessing them concurrently. |
| Timestamps | A **unique identifier** is generated for each transaction based upon transaction **start time**. |
| Optimistic Protocols | Based upon **validation** of the transaction after it executes its operations. |

### Locking Protocols

A **lock** is a variable associated with a data item:

|  |
| --- |
| A **binary lock** can have only two states - **locked** & **unlocked**.  If a binary lock is on a data item being requested, the transaction must **wait** until it is unlocked. |
| A **read/write lock** is used when multiple transactions want to **read** a data item concurrently.  If a transaction wants to **write** an item, then it must have **exclusive** access.  Some DBMSs allow **lock conversion** by upgrading to write lock, or downgrading to read lock. |

### Two-Phase Locking

A transaction follows **two-phase locking** if all lock operations precede the first unlock operations in that transactions. It is done to guarantee serializability.

**Expanding phase**: New locks on items can be acquired, but none can be released.

**Shrinking phase**: Existing locks can be released, but no new locks are obtainable.

If lock conversion is allowed, upgrading must occur in the expanding phase, and downgrading in the shrinking phase.

If every transaction in a schedule follows the two-phase locking protocol, the schedule is **serializable**.

*Limitation:* Can limit transaction concurrency if a transaction is not able to release X before acquiring Y. Other transactions accessing X must wait.

### Deadlock

**Deadlock** is when each transaction in the set is **blocked** i.e. waiting for one of the other transactions to release a lock.

**Deadlock prevention**: TA is waiting for TB to release a lock.

|  |  |
| --- | --- |
| No waiting | TA is aborted and **restarted** after a certain delay. |
| Wait-die | If TA is older than TB, then TA is allowed to **wait**. Otherwise, abort TA and **restart** later with the same timestamp. |
| Wound-wait  (Opposite to wait-die) | If TA is older than TB, then TB is aborted and **restarted** later with the same timestamp.  Otherwise, TA is allowed to **wait**. |
| Cautious waiting | If TB is not blocked, then TA is allowed to **wait**. Otherwise **abort** TA. |

**Deadlock detection**:

|  |
| --- |
| Construct and maintain a **wait-for graph**.   * One node is created for each transaction currently executing. * When a transaction is **blocked**, a directed edge to the transaction with the lock on the data is created. * When the lock is **released**, the directed edge is dropped. * Deadlock is occurring if a **cycle** exists within the graph. |

When deadlock is detected, one of the transactions must be aborted and rolled back.

**Victim selection** is used to select the cache to be aborted:

* Avoid transactions that have run for a long time / performed many updates.
* Select transactions that have not made many changes.

### Starvation

**Starvation** is when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. Can occur if the waiting scheme gives some transactions priority or from victim selection.

**Starvation solutions**:

|  |
| --- |
| **First-Come-First-Served Queue**:  Transactions are allowed to lock an item in the order in which they originally requested it. |
| **Waiting List**:  Allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it gets the highest priority and proceeds. |
| **Victim Priority**:  Assigns higher priorities to transactions that have been aborted multiple times. |

### Timestamp Ordering

For each item accessed by conflicting operations in the schedule, the order in which the item is accessed must not violate timestamp ordering.

The DBMS keeps two timestamp values for each data item:

1. read\_TS(X)  
   Timestamp of the most recently **started** transaction that has successfully **read** item X.
2. write\_TS(X)  
   Timestamp of the most recently **started** transaction that has successfully **written** item X.

Whenever a transaction T issues a:

|  |  |
| --- | --- |
| write\_item(X) | if (read\_TS(X) > TS(T)) or (write\_TS(X) > TS(T)) then:   * A younger transaction has already read / written to X before T has written to it, thus violating timestamp ordering. * Abort and rollback T.   if (read\_TS(X) ≤ TS(T)) and (write\_TS(X) ≤ TS(T)) then:   * Execute operation. * Set write\_TS(X) = TS(T). |
| read\_item(X) | if (write\_TS(X) > TS(T)) then:   * A younger transaction has already written to X before T has read it, thus violating timestamp ordering. * Abort and rollback T.   if (write\_TS(X) ≤ TS(T)) then:   * Execute operation. * Set read\_TS(X) to the maximum of TS(T) and read\_TS(X). |