TWO PHASE LOCKING TECHNIQUE

Types of Locks and System Lock Tables

Locks are used in database systems to manage access to data when multiple transactions are running at the same time. Different types of locks exist to help control concurrency (simultaneous access).

Binary Locks

A **binary lock** has only two states:

1. **Locked (1)** – The data item is unavailable to other transactions.
2. **Unlocked (0)** – The data item is available for access.

Each database item **X** has its own lock. If the lock value is **1**, other transactions cannot access it. If the lock value is **0**, the transaction can access the item but must **change the lock value to 1** before using it.

The two operations used with binary locks are:

- **lock_item(X)** – A transaction must request a lock before accessing a data item. If the item is locked (LOCK(X) = 1), the transaction must wait. If unlocked (LOCK(X) = 0), the transaction sets LOCK(X) to 1 and proceeds.
- **unlock_item(X)** – Once the transaction is done using the item, it releases the lock by setting LOCK(X) back to 0, allowing other transactions to access it.

How Binary Locks Work

1. When a transaction requests a lock on item **X**, it checks whether LOCK(X) = 0.
2. If LOCK(X) = 0, it sets LOCK(X) = 1 and accesses the item.
3. If LOCK(X) = 1, the transaction **waits** until the lock is released.
4. When the transaction finishes using **X**, it calls **unlock_item(X)** to set LOCK(X) = 0.
5. If there are waiting transactions, one of them is allowed to proceed.

These operations must be **atomic**, meaning that once started, they cannot be interrupted until they finish. The lock manager in the **Database Management System (DBMS)** keeps track of which transactions hold locks and manages access.

Rules for Using Binary Locks

1. A transaction **must lock an item before** reading or writing it.
2. A transaction **must unlock an item after** finishing with it.
3. A transaction **cannot lock an item if it already holds the lock**.
4. A transaction **cannot unlock an item it does not hold a lock on**.

The system stores locked items in a **lock table**, which helps the **lock manager** control access.

Shared/Exclusive (Read/Write) Locks

Binary locks are **too restrictive** because only **one transaction can access an item at a time**. In practice, **multiple transactions** should be able to read the same item simultaneously.

Shared/Exclusive (or Read/Write) locks solve this problem by allowing:

- **Shared locks (read-locks)** – Multiple transactions can **read** the same item at the same time.
- **Exclusive locks (write-locks)** – Only one transaction can **write** to an item, preventing others from reading or writing it.

These locks are implemented using **three operations**:

1. **read_lock(X)** – Grants shared access to **X**. Multiple transactions can hold a read lock on **X** simultaneously.
2. **write_lock(X)** – Grants exclusive access to **X**. No other transaction can read or write **X** until the lock is released.
3. **unlock(X)** – Releases the lock so other transactions can access **X**.

How Shared/Exclusive Locks Work

- If a transaction **requests a read lock** on **X**, it is granted if X is currently **unlocked** or already read-locked.
- If a transaction **requests a write lock**, it must wait until **X is completely unlocked**.
- If a transaction **holding a read lock wants to write**, it must **upgrade** its lock to a write lock.
- If a transaction **holding a write lock** wants to downgrade it to a read lock, it can do so.

Each **lock table** record contains:

1. **Data item name**
2. **LOCK status** (read-locked or write-locked)
3. **Number of transactions** holding a read lock
4. **List of transactions** holding the lock

These operations should also be **atomic**, meaning they should not be interrupted until they are completed.

Rules for Shared/Exclusive Locks

1. A transaction must **request a lock before reading or writing** an item.
2. A transaction must **request a write lock before modifying** an item.
3. A transaction must **release the lock** after finishing its operations.
4. A transaction **cannot request a lock if it already holds one**.
5. A transaction **cannot unlock an item it does not hold a lock on**.

Lock Conversion (Upgrading & Downgrading)

Sometimes, a transaction **needs to change its lock type**:

- **Upgrading a lock:** If a transaction holds a **read lock** but later needs to **write**, it can **upgrade** to a write lock.
    - This upgrade is **only allowed** if the transaction is the **only one holding a read lock**.
- **Downgrading a lock:** If a transaction holds a **write lock** but only needs to **read**, it can **downgrade** to a read lock.

To support **lock conversion**, the lock table must store the **transaction ID** of the lock holder.