**🔍 What is a Shared Lock (S-Lock)?**

A **Shared Lock** is used when a transaction **wants to read** a data item but **not modify** it.

* Multiple transactions **can hold shared locks simultaneously** on the same data item.
* No transaction can **write/update** the data item as long as a shared lock is held.

It **ensures consistency** by preventing dirty reads and lost updates, but allows **read concurrency**.

**✅ Rules of Shared Lock:**

1. **Multiple S-Locks allowed** on the same data item (multiple reads).
2. **No X-Lock** (write lock) is allowed on the same data item until all S-Locks are released.
3. A transaction must release its **S-Lock** after completing the read operation.

**💡 Real-Life Analogy**

Think of a **library book**:

* Multiple people can **read the same book** in the reading room.
* But **no one can write/mark** in the book while others are reading it.

Assume we have a database table:

Employee

---------

ID | Name | Salary

---|--------|--------

1 | Alice | 50000

2 | Bob | 60000

Let’s consider **two transactions: T1 and T2**

Step 1: Transaction T1 begins and wants to read Alice's salary

-- T1

BEGIN;

SELECT Salary FROM Employee WHERE ID = 1;

-- Acquires Shared Lock (S-Lock) on Employee[ID=1]

DB Engine applies S-Lock on Employee[ID=1] for T1

Step 2: Transaction T2 also wants to read the same record

-- T2

BEGIN;

SELECT Name FROM Employee WHERE ID = 1;

-- Also acquires Shared Lock (S-Lock) on Employee[ID=1]

**Allowed!** Since both are just reading, the lock system **permits** this.

Step 3: Transaction T3 wants to **update** Alice’s salary

-- T3

BEGIN;

UPDATE Employee SET Salary = 55000 WHERE ID = 1;

-- Requests Exclusive Lock (X-Lock) on Employee[ID=1]

**Blocked!**  
Because **T1 and T2 hold S-Locks**, the database will **not grant the X-Lock** to T3.  
T3 must **wait** until both T1 and T2 release their S-Locks.

Step 4: T1 and T2 finish reading

-- T1

COMMIT;

-- Releases S-Lock

-- T2

COMMIT;

-- Releases S-Lock

Now that the **S-Locks are released**, the **X-Lock can be granted** to T3.

Step 5: T3 updates the record

-- T3

UPDATE Employee SET Salary = 55000 WHERE ID = 1;

COMMIT;

-- X-Lock acquired and released after commit

A screenshot of a computer

AI-generated content may be incorrect.

**✅ Advantages of Shared Lock (S-Lock)**

| **Advantage** | **Description** |
| --- | --- |
| 🔄 **Supports Concurrent Reads** | Multiple transactions can **read** the same data item **at the same time**, improving **read performance**. |
| 🔐 **Prevents Dirty Reads** | Ensures that a transaction does **not read uncommitted data** modified by another transaction. |
| 📊 **Improves Throughput for Read-Heavy Workloads** | Ideal for systems where **reading is more frequent than writing**, such as reporting or analytics systems. |
| ⚖️ **Helps Maintain Isolation** | Helps ensure **ACID isolation** by preventing interference from uncommitted writes. |
| 🔍 **Low Overhead Compared to Exclusive Locks** | Since multiple readers are allowed, it’s less restrictive than exclusive locks. |

**❌ Disadvantages of Shared Lock (S-Lock)**

| **Disadvantage** | **Description** |
| --- | --- |
| ⛔ **Blocks Writers (X-Locks)** | While a Shared Lock is held, **no transaction can write** to the data item, even if readers don’t change the data. |
| 🔄 **Can Lead to Writer Starvation** | In read-heavy systems, continuous shared locks might prevent writers from ever acquiring exclusive locks. |
| ⚠️ **Can Cause Deadlocks** | When combined with other types of locks, improper lock management can lead to **deadlocks**. |
| 🕒 **Increases Wait Time for Writers** | Writers must **wait** for all shared locks to be released before acquiring an exclusive lock. |
| 🧠 **Requires Careful Management** | Misuse or long-held S-Locks can create **performance bottlenecks** or **reduce availability** for update operations. |

**🧪 Real-Life Example Recap**

* Two transactions (T1, T2) read the same row using S-Lock: ✅ OK
* A third transaction (T3) tries to **update** the same row: ❌ BLOCKED until S-Locks are released

**📝 Summary Table**

| **Criteria** | **Shared Lock (S-Lock)** |
| --- | --- |
| Allows multiple readers? | ✅ Yes |
| Allows writers concurrently? | ❌ No (writers are blocked) |
| Use case | Safe concurrent **reads** |
| Risk | Writer **starvation**, **deadlocks** |
| Best for | Read-heavy applications |

**🔒 What is an Exclusive Lock (X-Lock)?**

An **Exclusive Lock** (also known as a **Write Lock**) is a type of lock used in database concurrency control to ensure that **only one transaction can read and write** a data item at a time.

**✅ Key Characteristics of Exclusive Lock:**

| **Feature** | **Exclusive Lock (X-Lock)** |
| --- | --- |
| Allows reading by others? | ❌ No |
| Allows writing by others? | ❌ No |
| Allows same transaction to write? | ✅ Yes |
| Prevents | Lost updates, dirty reads/writes |

**✅ Rules of Exclusive Lock**

1. **Only one transaction** can hold an X-Lock on a data item.
2. While an X-Lock is held:
   * No **other transaction** can read (**S-Lock**) or write (**X-Lock**) the same item.
3. A transaction must **acquire an X-Lock** before **updating or deleting** a data item.

**📘 Real-Life Analogy**

Think of editing a **shared Google Doc**:

* When someone is editing a paragraph, others cannot read or edit that paragraph until the person finishes.
* Prevents **conflicting changes**.

Suppose we have a table:

Accounts

---------

AccountID | Balance

----------|--------

101 | 5000

102 | 3000

Two transactions, **T1** and **T2**, want to update the same account balance.

Step 1: Transaction T1 wants to update Account 101's balance

-- T1

BEGIN;

UPDATE Accounts SET Balance = Balance - 1000 WHERE AccountID = 101;

👉 The database engine will:

* Acquire an **Exclusive Lock (X-Lock)** on Accounts[AccountID=101] for T1
* Proceed with the update
* Prevent any other transaction from reading or writing AccountID = 101

Step 2: Transaction T2 also tries to read the same account

-- T2

BEGIN;

SELECT Balance FROM Accounts WHERE AccountID = 101;

🔒 **Blocked!**  
T2 cannot even read the value because T1 is holding an X-Lock.

❌ Step 3: Transaction T2 tries to update the same record

-- T2

UPDATE Accounts SET Balance = Balance + 1000 WHERE AccountID = 101;

🔒 **Still blocked!**  
T2 must wait for T1 to **release the X-Lock**.

Step 4: Transaction T1 completes

-- T1

COMMIT;

-- Releases X-Lock on AccountID = 101

Step 5: Now T2 can proceed

-- T2

UPDATE Accounts SET Balance = Balance + 1000 WHERE AccountID = 101;

-- X-Lock granted to T2 now

COMMIT;

A screenshot of a computer

AI-generated content may be incorrect.

**🟩 Advantages of Exclusive Lock**

* **Ensures data consistency** during write operations.
* Prevents **lost updates** and **dirty reads/writes**.
* Guarantees **atomicity** of update/delete operations.

**🟥 Disadvantages of Exclusive Lock**

* Reduces **concurrency** (no other read/write allowed).
* Can lead to **deadlocks** if multiple transactions wait on each other.
* May cause **performance bottlenecks** under high write loads.

**Two-Phase Locking Protocol (2PL)** is a fundamental concurrency control mechanism used in databases to ensure **serializability** — that is, the outcome of transactions is the same as if the transactions were executed one after another in some serial order.

**📌 What is Two-Phase Locking (2PL)?**

The **Two-Phase Locking protocol** ensures that all **conflicting operations** (like read/write on same data) are **controlled using locks**, and that the **order of lock and unlock operations** is strictly followed to maintain consistency.

**✅ Two Phases of 2PL:**

1. **Growing Phase**:
   * A transaction **may acquire locks** (shared or exclusive).
   * It **cannot release** any lock.
   * This continues until the transaction reaches its **lock point** (last lock acquisition).
2. **Shrinking Phase**:
   * A transaction **may release locks**.
   * It **cannot acquire** any new locks.

❗ Once a transaction **releases** its first lock, it enters the shrinking phase and **cannot acquire** any new locks.

Time →

┌───────────────┬───────────────┐

│ Growing Phase │ Shrinking Phase │

└───────────────┴───────────────┘

Acquire Locks Release Locks Only

**How 2PL Works**

**Scenario: We have two transactions T1 and T2, and a table Accounts:**

Accounts

---------

ID | Balance

101 | 1000

102 | 2000

**Transaction T1:**

-- T1

Read Balance from Account 101

Write new Balance to Account 101

Transaction T2:

-- T2

Read Balance from Account 101

**🔄 Step-by-step 2PL Execution:**

**Step 1: T1 starts – GROWING phase**

* T1 requests **Exclusive Lock (X-Lock)** on Account 101 (for reading and updating).
* Lock granted.
* T1 reads and updates balance.
* ✅ Still in **growing phase**.

**Step 2: T2 starts – GROWING phase**

* T2 requests **Shared Lock (S-Lock)** on Account 101.
* ❌ **Blocked**, because T1 holds an exclusive lock.

**Step 3: T1 finishes – SHRINKING phase**

* T1 **commits** and **releases the lock**.
* T1 now in **shrinking phase** — cannot acquire new locks.

**Step 4: T2 resumes**

* T2 now **acquires S-Lock** on Account 101.
* T2 reads balance.
* T2 releases S-Lock on commit.

✅ **Serializability** is ensured because:

* T1 completed all its lock operations before releasing any.
* T2 could not interfere while T1 was running.

**⚖️ Advantages of 2PL**

| **Advantage** | **Explanation** |
| --- | --- |
| ✅ **Ensures Serializability** | Prevents conflicting operations from breaking consistency rules. |
| ✅ **Simple to Implement** | Clear locking rules make it straightforward to enforce. |
| ✅ **Works with Shared and Exclusive Locks** | Ensures safe reads and writes. |

**⚠️ Disadvantages of 2PL**

| **Disadvantage** | **Explanation** |
| --- | --- |
| 🌀 **Deadlocks Possible** | Two transactions waiting on each other's locks can cause deadlock. |
| 🧍‍♂️ **Blocking Delays** | Transactions may be blocked waiting for locks to be released. |
| 🧠 **Overhead** | Managing locks and lock tables consumes system resources. |
| 🚫 **Reduced Concurrency** | Restrictive locking can slow down performance in high-volume systems. |

**🧠 Deadlock Example under 2PL:**

| **Transaction** | **Step** |
| --- | --- |
| T1: X-Lock on A ✅ |  |
| T2: X-Lock on B ✅ |  |
| T1: wants X-Lock on B ❌ (Blocked by T2) |  |
| T2: wants X-Lock on A ❌ (Blocked by T1) |  |

🔄 **Deadlock** — both wait forever unless a timeout or deadlock detection mechanism breaks the cycle.

**🧩 Types of Two-Phase Locking**

| **Type** | **Description** |
| --- | --- |
| **Strict 2PL** | All locks held until transaction commits or aborts. Prevents cascading aborts. |
| **Rigorous 2PL** | Same as strict, but **even read locks** are held until commit. |
| **Conservative (Static) 2PL** | Transaction gets **all locks at once** before it starts. Avoids deadlocks. |

**📝 Summary**

| **Concept** | **Details** |
| --- | --- |
| 2PL Phases | Growing (acquire locks), Shrinking (release locks) |
| Guarantees | Serializability |
| Risk | Deadlocks, blocking |
| Use Case | Multi-user database systems with consistency needs |

A diagram of a lock system

AI-generated content may be incorrect.

**✅ What Is Time-Based Concurrency Control?**

It is a method used in databases to make sure that multiple users (or **transactions**) accessing data at the same time do **not mess up the data**.

Instead of using **locks**, it uses **timestamps** to decide the **order** in which transactions should happen.

**🕰️ What Is a Timestamp?**

A **timestamp** is just a number (usually based on the time when a transaction starts).  
It helps the database decide **who is older and who is newer**.

Example:

* Transaction **T1** starts at 10:01 → timestamp = 1
* Transaction **T2** starts at 10:02 → timestamp = 2  
  So, **T1 is older** than T2.

**📦 How It Works (Simple Rules):**

Let’s say you have a data item X.

Each data item keeps track of:

* WTS(X): Timestamp of the last transaction that **wrote** to X
* RTS(X): Timestamp of the last transaction that **read** from X

Now suppose a new transaction T comes in:

1. ✅ If T wants to **read(X)**:
   * It can read **only if** T is newer than the last write.
   * If T is **older than WTS(X)** → ❌ **abort** (you’re trying to read old/stale data).
2. ✅ If T wants to **write(X)**:
   * It can write **only if**:
     + T is newer than the last read (RTS(X))
     + T is newer than the last write (WTS(X))
   * If not → ❌ **abort** (you’re trying to overwrite data that newer transactions already read or wrote).

**🧪 Example: Easy to Understand**

Let’s say:

text

CopyEdit

Data item X = 100

WTS(X) = 3 → last written by T3

RTS(X) = 4 → last read by T4

**Transactions:**

| **Transaction** | **Timestamp** |
| --- | --- |
| T1 | 2 |
| T5 | 5 |

**🔹 T1 tries to READ X**

* T1's timestamp = 2
* But WTS(X) = 3 → T1 is **older than the last write**
* ❌ T1 is trying to read **stale data** → **ABORT**

**🔹 T5 tries to READ X**

* T5’s timestamp = 5
* WTS(X) = 3 → T5 is newer → ✅ **READ allowed**
* Update RTS(X) = 5

**🔹 T1 tries to WRITE X**

* T1’s timestamp = 2
* RTS(X) = 5 → T1 is **older than the last read**
* ❌ T1 cannot write → **ABORT**

**🔹 T5 tries to WRITE X**

* T5’s timestamp = 5
* WTS(X) = 3, RTS(X) = 5 → both are ≤ T5
* ✅ T5 is newer → **WRITE allowed**
* Update WTS(X) = 5

**✅ Advantages (Simple View)**

| **Advantage** | **Why It's Good** |
| --- | --- |
| No Locking | Doesn’t use locks → no deadlocks |
| Always Consistent | Makes sure data is in correct order |
| Fast Access | Good for systems with lots of reads/writes |

**❌ Disadvantages**

| **Disadvantage** | **Why It's Bad** |
| --- | --- |
| More Aborts | Old transactions may be rejected a lot |
| Waste of Effort | Transactions may do work and then get aborted |
| Starvation | Some transactions may never finish |

**🎯 Real-Life Analogy**

Think of transactions like people entering a library to write in a shared notebook:

* Everyone gets a **number** when they walk in (timestamp).
* You are **not allowed** to write or read if someone **newer than you has already done it**.
* The library ensures that **older people don't overwrite or read outdated things** written by newer people