The two main approaches are to guaranteeing conflict-serializability are:

locking protocols

time-stamping

Locking

A *lock* is a variable associated with a data item in the database is used to signify the status of that variable wrt to possible access to the item.

Binary Locks:

A binary lock may have two states: locked or unlocked. If an item is locked by one transaction it cannot be accessed by another transaction

Transaction must be well-formed.

For a transaction `T` to gain any access to a database item `x`, `T` must first issue a `lock(x)` request.

If this request is successful, `T` can access `X`. Otherwise it must wait.

Having completed accessing item `X`, `T` must issue an `unlock(x)` to free up the database item.

Must not unlock free item, or attempt to lock an item it has already locked.

Consider:

```
T1                                  T2

read_item(X);
 X := X + 50;

                                    read_item(X)
                                    X := X - 50;

write_item(X)

                                    write_item(X)
```

By applying a locking procedure to the above, T2's `read_item` is queued until T1 finishes and issues an `unlock(X)`

Causes a different ordering of accesses to the database items.

The binary lock approach is too restrictive for real-world applications.

Shared and Exclusive Locks

The `lock(X)` variable can be in one of three states:

read locked
write locked
unlocked

It an item is write locked, no other transaction can obtain a read or write lock on the database item.

If an item is read locked, no other transaction can attain a write lock.

Hence, we allow multiple readers but only one writer.

Transaction can issue `read_lock(x)`, `write_lock(x)` and `unlock(x)`.

Often augmented with `upgrade(x)` and `downgrade(x)`.

These locking schemes on their own do not guarantee serializability.

Two phase locking

They are used in conjunction with locking protocols to ensure correctness. The most commonly used protocol is the 2-phase locking protocol.

Two phase locking (2PL) states that a transaction cannot issue a lock request once an unlock request has been issued.

Hence, there are 2 phases: the growing phases (where transactions obtain locks) and the shrinking phase where the transactions release locks.

2PL locking guarantees conflict serializability.

Exercise: Apply 2PL to the example of the lost update problem and the temporary update problem.

Exercise: Prove that 2PL guarantees conflict serializability (hint: proof by contradiction).

Consider the following schedule:

```
    T1                  T2
write_lock(X)
write_item(X)
                    write_lock(Y)
                    write_item(Y)
write_lock(Y)

/* queued */
                  write_lock(X)
                   /* queued */
```

-> Deadlock

We need some means to handle deadlock.

We need to first detect deadlock and then resolve the deadlock.

Usually, the lowest priority transaction is terminated. Other transactions can then continue.

Deadlock detection usually involves the creation of a dependency graph. If a cycle exists in the dependency graph, deadlock exists.

Usually triggered by a transaction that has been queued for a certain period of time.

Two-Phase Locking Techniques:

Different variations exist.

Basic: Transaction locks data items incrementally. This may cause deadlock.

Conservative:

Prevents deadlock by locking all desired data items before transaction begins execution.

Strict: A stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm

LOCK(X):

```
B:  if LOCK (X) = 0          //item is unlocked
    then LOCK (X) ← 1  //lock the item
    else begin
              wait (until lock (X) = 0) and
              the lock manager wakes up the transaction);
    goto B
    end;
```

UNLOCK(X):

```
     LOCK (X) ← 0
    if any transactions are waiting for X,
              wake up a waiting transaction.
```

Shared and Exclusive Locks:

read_lock(X)

```
B: if LOCK (X) = "unlocked" then
        begin LOCK (X) ← "read-locked";
              no_of_reads (X) ← 1;
        end
    else
        if LOCK (X) ← "read-locked" then
              no_of_reads (X) ← no_of_reads (X) +1
         else
              begin wait (until LOCK (X) = "unlocked" and
                    the lock manager wakes up the transaction);
                go to B
              end;
```

write_lock(X):

```
B: if LOCK (X) = "unlocked" then
     begin LOCK (X) ← "write_locked";
          end
     else
          begin wait (until LOCK (X) = "unlocked" and
                    the lock manager wakes up the transaction);
                    go to B
          end;
```

unlock(X):

```
if LOCK (X) = "write-locked" then
    begin LOCK (X) ← "unlocked";
        wakes up one of the transactions, if any
    end
    else if LOCK (X) ← "read-locked" then
        begin
            no_of_reads (X) ← no_of_reads (X) -1
            if  no_of_reads (X) = 0 then
            begin
                LOCK (X) = "unlocked";
                wake up one of the transactions, if any
            end
        end;
```