CHAPTER – 14

CONCURRENCY

Prepared By: Prof. Kinjal Acharya
Assistant Professor
DDU, Nadiad

THREE CONCURRENCY PROBLEMS: (WITHOUT LOCKING)

- DEFINITION: The DBMSs allow many transactions to access the same database at the same time. This phenomenon is called Concurrency.
- Some kind of concurrency control mechanism is needed to ensure that concurrent transactions do not interfere with each other.
- There are essentially three ways, that is in which a transaction, through correct in itself, can produce the wrong answer if some other transaction might also be correct in itself.
- It is the uncontrolled interleaving of operations from the two correct transactions that produces the overall incorrect result.
- The three problems are:
  1. The lost update problem,
  2. The uncommitted dependency problem (or dirty read problem)
  3. The inconsistent analysis problem.

1. THE LOST UPDATE PROBLEM:

- Consider the situation illustrated below.
- Transaction A retrieves some tuple t at time t1; transaction B retrieves that same tuple t at time t2; transaction A updates the tuple (on the basis of the values seen at time t1) at time t3; and transaction B updates the same tuple (on the basis of the values seen at time t2, which are the same as those seen at time t1) at time t4.
- Transaction A's update is lost at time t4, because transaction B overwrites it without even looking at it.

THE LOST UPDATE PROBLEM: EXAMPLE

Transaction A	Time	Transaction B
-		-
-		-
RETRIEVE t	t1	-
-		-
-	t2	RETRIEVE t
-		-
UPDATE t	t3	-
-		-
-	t4	UPDATE t
-		-

THE UNCOMMITTED DEPENDENCY PROBLEM (OR DIRTY READ PROBLEM)

- The uncommitted dependency problem arises if one transaction is allowed to retrieve or update a tuple that has been updated by another transaction but not yet committed by that other transaction.
- If it has not yet been committed, there is always a possibility that it never will be committed but will be rolled back, in which case the first transaction will have seen some data that now no longer exists
- In the first example, transaction A sees an uncommitted update (also called an uncommitted change) at time t2. That update is then undone at time t3. Transactions A is therefore operating on a false assumption, the assumption that tuple t has the value a seen at time t2, whereas it has whatever value it had prior to time t1.
- As a result, transaction A might well produce incorrect output.

Example 1

Transaction A	Time	Transaction B
-		-
-	t1	UPDATE t
-		-
RETRIEVE t	t2	-
-		-
-	t3	ROLLBACK

THE UNCOMMITTED DEPENDENCY PROBLEM (OR DIRTY READ PROBLEM)

- The second example is even worse.
- Not only does transaction A become dependent on an uncommitted change at time t2, but it actually loses an update at time t3 – because the rollback at time t3 causes tuple t to be restored to its value prior to time t1.
- This is another version of the lost update problem.

Example 2

Transaction A	Time	Transaction B
-		-
-	t1	UPDATE t
-		-
UPDATE t	t2	-
-		-
-	t3	ROLLBACK
-		

THE INCONSISTENT ANALYSIS PROBLEM

- Consider two transactions A and B operating on account (ACC) tuples.
- Transaction A is summing account balances; transaction B is transferring an amount 10 from account 3 to account 1.
- The result produced by A, 110, is obviously incorrect; if A were to go on to write that result back into the database, it would actually leave the database in an inconsistent state.
- A has seen an inconsistent state of the database and has therefore performed an inconsistent analysis.

The initial values of accounts are given below:
ACC 1 = 40
ACC 2 = 50
ACC 3 = 30

Transaction A	Time	Transaction B
-		-
RETRIEVE ACC 1:	t1	-
Sum = 40		-
RETRIEVE ACC 2:	t2	-
Sum = 90		-
-	t3	RETRIEVE ACC 3
-		-
	t4	UPDATE ACC 3:
		30 → 20
-	t5	RETRIEVE ACC 1
-		-
-	t6	UPDATE ACC 1:
-		40 → 50
-	t7	COMMIT
-		
RETRIEVE ACC 3:	t8	
Sum = 110, not 120		

LOCKING

- The problems of concurrency can all be solved by means of a concurrency control technique called **locking**.
- When a transaction needs some tuple it is interested in, will acquire a lock on that.
- The effect of the lock is to "lock other transactions out of" the object, and thus in particular to prevent them from changing it.
- The first transaction is able to carry out its processing such that the object will remain in a stable state for as long as that transaction wishes it to.
- The system supports **two kinds of locks**:
  1. exclusive locks OR write locks (X locks) and
  2. shared locks OR read locks (S locks)
- If transaction A holds an exclusive (X) lock on tuple t, then a request from some distinct transaction B for a lock of either type on t will be denied.
- If transaction A holds a shared (S) lock on tuple t, then :
  1. A request from some distinct transaction B for an X lock on t will be denied;
  2. A request from some distinct transaction B for an S lock on t will be granted (that is, B will now also hold an S lock on t).

		A having locks	
		X	S
B Requesting Locks	X	N	N
	S	N	Y

DATA ACCESS PROTOCOL (OR LOCKING PROTOCOL)

- It makes use of X and S locks as just defined to guarantee that problems cannot occur.

1. A transaction that wishes to retrieve a tuple must first acquire as S lock on that tuple.
2. A transaction that wishes to update a tuple must first acquire an X lock on that tuple, Alternatively, if it already holds an S lock on the tuple (as it will in a RETRIEVE – UPDATE sequence), then it must promote that S lock to X level.
3. If a lock request from transaction B is denied because it conflicts with a lock already held by transaction A, transaction B goes into a wait state. B will wait until A's lock is released. The system must guarantee that B does not wait forever. A simple way to provide such a guarantee is to service all lock requests on a "first-come, first-served" basis.

- X locks are held until end-of-transaction (COMMIT or ROLLBACK). S locks are normally held until that time also.

THE THREE CONCURRENCY PROBLEMS WITH LOCKING

- Now we are going to solve the three concurrency problems by applying locks on tuples.
- The three problems:
    1. The lost update problem;
    2. The uncommitted dependency problem; and
    3. The inconsistent analysis problem

1. THE LOST UPDATE PROBLEM:

- Transaction A's UPDATE at time t3 is not accepted, because it is an implicit request for a X lock on t, and such a request conflicts with the S lock already held by transaction B.
- So A goes into a wait state. For analogous reasons, B goes into a wait state at time t4.
- Now both transactions are unable to proceed, so there is no question of any update being lost.
- Locking thus solves the lost update problem by reducing it to another problem, but at least it does solve the original problem.
- The new problem is called **deadlock**.

THE LOST UPDATE PROBLEM (WITH LOCKING): EXAMPLE

Transaction A	Time	Transaction B
-		-
-		-
RETRIEVE t	t1	-
(acquire S lock on t)		-
-	t2	RETRIEVE t
-		(acquire S lock on t)
UPDATE t	t3	
request x lock on t)		-
wait		
wait		-
wait		
wait	t4	UPDATE t
wait		(request X lock on t)
wait		wait
wait		wait

THE UNCOMMITED DEPENDENCY PROBLEM (WITH LOCKING)

- Transaction A's operation at time t2 is not accepted in either case, because it is an implicit request for a lock on t, and such a request conflicts with the X lock already held by B.
- So A goes into a wait state. It remains in that wait state until B reaches its termination (either COMMIT or ROLLBACK), when B's lock is released and A is able to proceed; and at that point A sees a committed value (either the pre-B value, if B terminates with rollback, or the post – B value otherwise).
- Either way, A is no longer dependent on an uncommitted update.

Transaction A	Time	Transaction B
-		-
-	t1	UPDATE t
-		(acquire X lock on t)
RETRIEVE t	t2	-
(request S lock on t)		-
wait	t3	COMMIT / ROLLBACK
Wait		(release X lock on t)
Resume: RETRIEVE t	t4	
(Acquire S lock on t)		

THE UNCOMMITED DEPENDENCY PROBLEM (WITH LOCKING)

Transaction A	Time	Transaction B
-		-
-	t1	UPDATE t
-		(acquire X lock on t)
UPDATE t	t2	-
(request X lock on t)		-
Wait		
Wait	t3	COMMIT / ROLLBACK
		(Release X locks on t)
Resume: UPDATE t	t4	
(Acquire X locks on t)		
-		

THE INCONSISTENT ANALYSIS PROBLEM (WITH LOCKING)

- Transaction B's UPDATE at time t6 is not accepted, because it is an implicit request for an X lock an ACC 1, and such a request conflicts with the S lock already held by A; so B goes into a wait state. Likewise, transaction A's RETRIEVE at time t7 is also not accepted, because it is an implicit request for an S lock an ACC3, and such a request conflicts with the X lock already held by B; so A goes into a wait state also. Again, therefore, locking solves the original problem (the inconsistent analysis problem, in this case) by forcing a deadlock.

ACC 1 = 40
ACC 2 = 50
ACC 3 = 30

Transaction A	Time	Transaction B
-		-
RETRIEVE ACC 1:	t1	-
(Acquire S lock on ACC 1)		-
Sum = 40		-
RETRIEVE ACC 2:	t2	-
(Acquire S lock on ACC 2)		-
Sum = 90		-
-	t3	RETRIEVE ACC 3
-		(acquire S lock on ACC 3)
-		-
-	t4	UPDATE ACC 3:
-		(acquire X lock on ACC 3)
-		30 → 20
-		-
-	t5	RETRIEVE ACC 1
-		(acquire S lock on ACC 1)
-		-
-	t6	UPDATE ACC 1:
-		(request X lock on ACC 3)
RETRIEVE ACC 3:		wait
(acquire S lock on ACC 3)	t7	wait
Wait		wait

DEADLOCK

- Locking can be used to solve the three basic problems of concurrency.
- Locking can introduce problems of deadlock.
- **Definition**: Deadlock is a situation in which two or more transactions are in a simultaneous wait state, each of them waiting for one of the others to release a lock before it can proceed.
- Following figure shows deadlock involving two transactions, but deadlocks may involves more number of transactions as well.
- If a deadlock occurs, it is desirable that the system detects it and breaks it.
- Two methods are used to break deadlock situation called **deadlock detection** and **deadlock avoidance**.

Transaction A	Time	Transaction B
-		-
-		-
LOCK r1 EXCLUSIVE	t1	-
-		-
-	t2	LOCK r2 EXCLUSIVE
-		-
LOCK r2 EXCLUSIVE	t3	-
wait		-
wait	t4	LOCK r1 EXCLUSIVE
wait		wait
wait		wait

DEADLOCK DETECTION

- Detecting the deadlock involves detecting a cycle in the **Wait-for Graph** (i.e., the graph of "who is waiting for whom").
- **VICTIM:**

  Breaking the deadlock involves choosing one of the deadlocked transactions – i.e., one of the transactions in the cycle in the graph as the **victim** and rolling it back, thereby releasing its locks and so allowing some other transaction to proceed.
- Not all systems do in fact detect deadlocks; some just use a timeout mechanism and simply assume that a transaction that has done no work for some prescribed period of time is deadlocked.
- The victim has "failed" and been rolled back through no fault of its own.
- Some systems will automatically restart such a transaction from the beginning, on the assumption that the conditions that caused the deadlock in the first place will probably not arise again.
- Other system simply send a "deadlock victim" exception code back to the application; it is then up to the program to deal with situation in some graceful manner.

DEADLOCK AVOIDANCE

- Instead of allowing deadlock to occur and deal with them, it is possible to avoid deadlock entirely.
- To avoid we need to modify locking protocol in various ways.
- Let us consider one possible approach here.
- This approach comes in two versions, called Wait-Die and Wound-Wait.
- It works as follow:
- Every transaction is *timestamped* with its start time(unique time).
- When transaction A requests a lock on tuple that is already locked by transaction B, then:
- **Wait-Die:** 'A 'wait if it is older than B, otherwise it "dies" – that is, A is rolled back and restarted.
- **Wound-Wait:** 'A' waits if it is younger than B, otherwise it "wounds" B – that is, B is rolled back and restarted.
- If a transaction has to be restarted, it retains its original timestamp.

SERIALIZABILITY

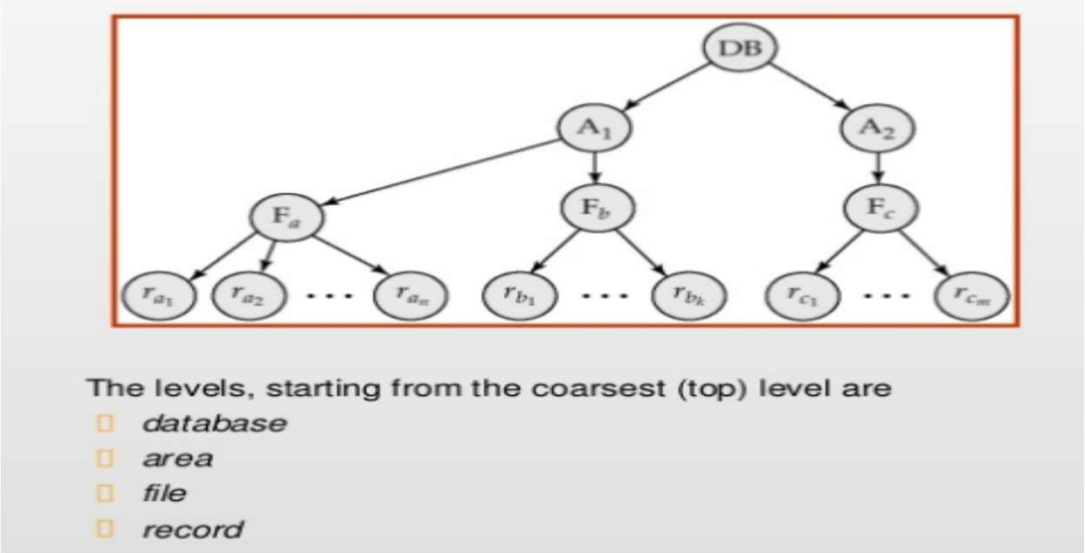
- **Serializability** is criterion for correctness for the execution of a given set of transactions.
- A given execution of a transactions is considered to be correct if it is serializable – i.e. if it produces the same result as some serial execution of the same transactions, running them one at a time.
- Individual transactions are assumed to be correct – i.e., they are assumed to transform a correct state of the database into another correct state.
- Running the transactions one at a time in any serial order is therefore also correct – "any" serial order because individual transactions are assumed to be independent of on another.
- The **transactions are interleaved**, means second transaction can be started before the first one could end. And execution can switch between the transactions back and forth.
- It can also switch between multiple transactions.
- An interleaved execution is only correct if it is equivalent to serial execution of same set of transactions – i.e., if it is serializable.
- **Schedule:** A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of instructions/tasks.
- **Serial Schedule:** It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.
- **Interleaved schedule:** A schedule that is not serial is a interleaved schedule (or simply a non serial schedule).
- **Equivalent Schedules:** Two schedules are said to be equivalent if they are guaranteed to produce the same result, independent of the initial state of the database. Thus, a schedule is correct (i.e., serializable) if it is equivalent to some serial schedule.

SERIALIZABILITY

- Two different serial schedules involving the same set of transactions might well produce different results, and so that two different interleaved schedules involving those transactions might also produce different results and yet both be considered correct.
- For example, suppose transaction A is of the form “Add 1 to x” and transaction B is of the form “Double x” (where x is some item in the database).
- Suppose also that the initial value of x is 10.
- Then the serial schedule A then B gives x = 22, whereas the serial schedule B then A gives x = 21.
- These two results are equally correct, and any schedule that is guaranteed to be equivalent to either A then B or B then A is like wise correct.
- We have discussed:
  - A schedule is the order in which the operations of multiple transactions appear for execution.
  - Serial schedules are always consistent.
  - Non-serial schedules are not always consistent.
  - Some non-serial schedules may lead to inconsistency of the database.
  - Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.

INTENT LOCKING

- Locks should be applied to larger or smaller units of data – an entire relation variable (relvar), or even the entire database, or (going to the opposite extreme) a specific component within a specific tuple.
- Suppose some transaction T requests an X lock on some relvar R.
- On receipt of T's request, the system must check whether any other transaction already has a lock on any tuple R – for if it does, then T's request cannot be granted at this time.
- How can the system detect such a conflict? It is obviously undesirable to have to examine every tuple in R to see whether any of them is locked by any other transaction, or to have to examine every existing lock to see whether any of them is for a tuple in R.
- So we introduce another protocol, the intent locking protocol**.**
- **The Intent Locking Protocol: No transaction is allowed to acquire a lock on a tuple before first acquiring a lock – probably an intent lock on the relvar that contain it.**



LOCKING GRANULARITY

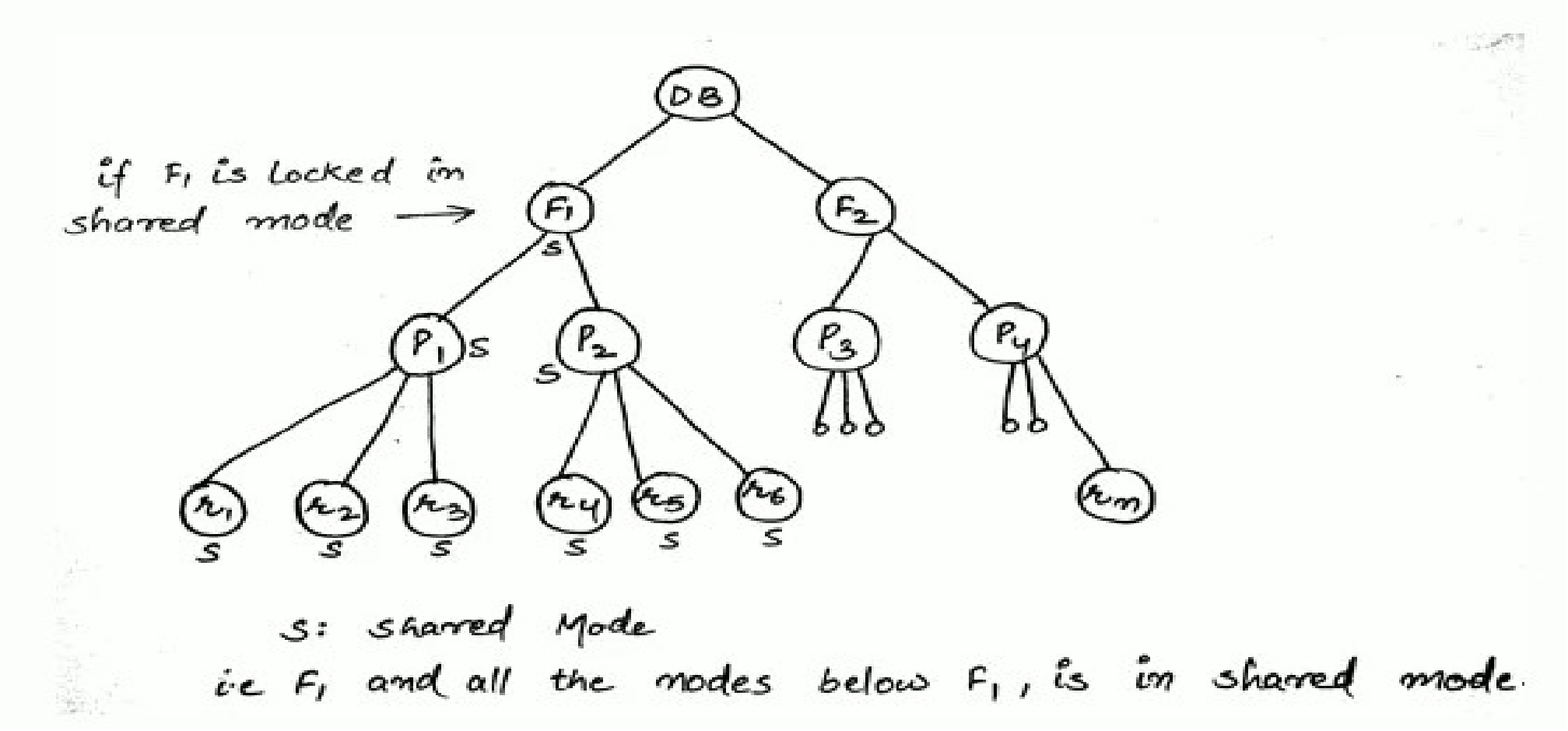
- The granularity of locks in a database refers to how much of the data is locked at one time. In theory, a database server can lock as much as the entire database or as little as one column of data. Such extremes affect the concurrency (number of users that can access the data) and locking overhead (amount of work to process lock requests) in the server.
- The granularity is a measure of the amount of data the lock is protecting.
- The finer the granularity, the greater the concurrency; the coarser, the fewer locks need to be set and tested and the lower the overhead.
- For example if a transaction has an X lock on an entire relvar, there is no need to set X locks on individual tuples within that relvar; on the other hand, no concurrent transaction will be able to obtain any locks on that relvar, or on tuples within that relvar, at all.
- Conflict detection in the example then becomes a comparatively simple matter of seeing whether any transaction has a conflicting lock at the relvar level.
- X and S locks make sense for whole relvars as well as for individual tuples.
- We now introduce three new kinds of locks, called intent locks that are for relvars, but not for individual tuples.
- The new kinds of locks are called intent shared (IS), intent exclusive (IX), and shared intent exclusive (SIX) locks, respectively.

INTENT LOCKS

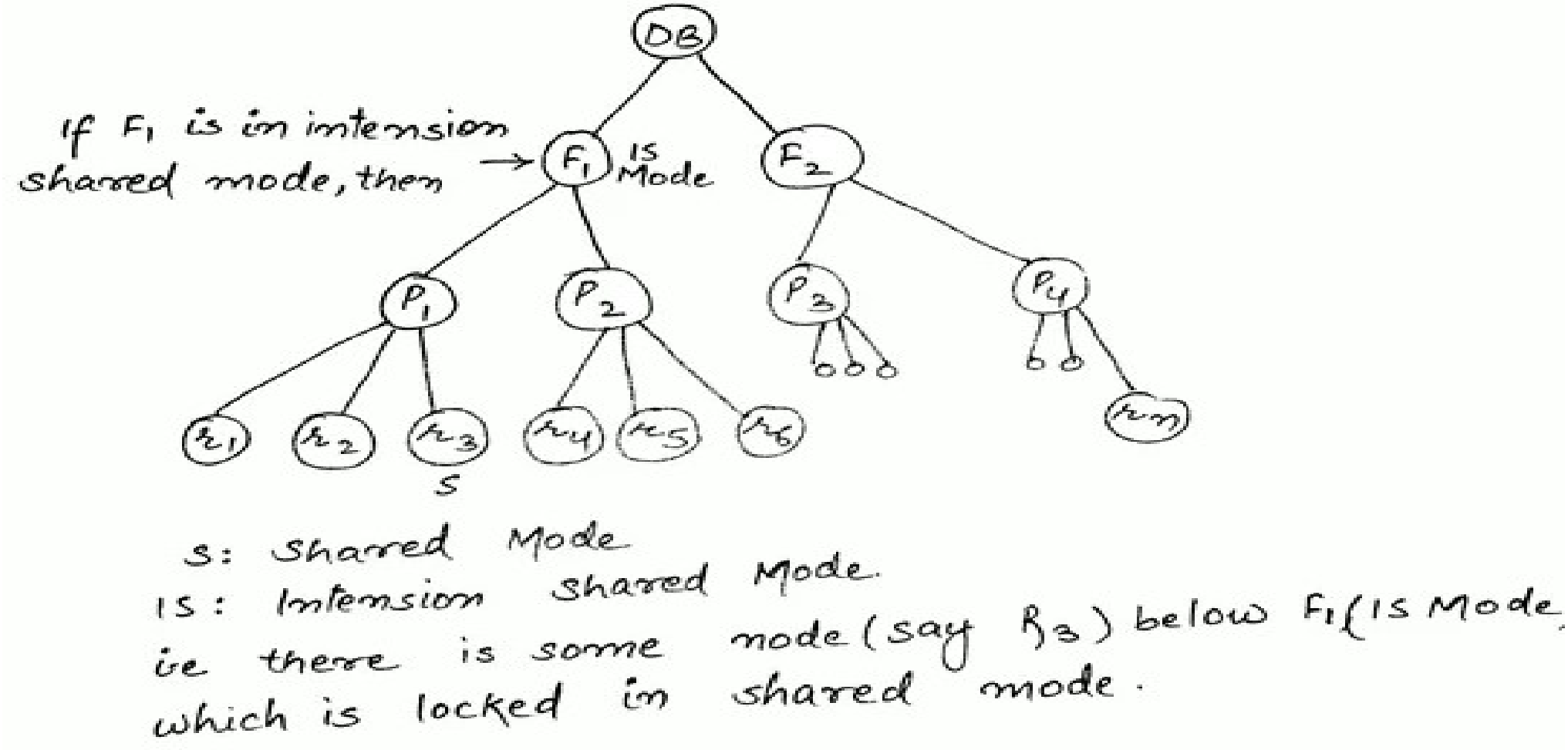
- **Intent Shared(IS) :** T intends to set S lock on individual tuples in R, in order to guarantee the stability of those tuples while they are processed.
- **Intent Exclusive(IX) :** Same as IS, plus T might update individual tuple in R and therefore set X locks on those tuples.
- **Shared(S) :** T can tolerate concurrent readers, but not concurrent updaters in R.
- **Shared Intent Exclusive(SIX) :** Combines S and IX, that is T can tolerate concurrent readers, but not concurrent updaters in R plus T might update individual tuple in R and therefore set X locks on those tuples.
- **Exclusive(X) :** T cannot tolerate any concurrent access to R at all.

```
S   : Shared Locks/Mode
X   : Exclusive Locks/Mode
IS  : Intention Shared Locks/Mode
IX  : Intention Exclusive Locks/Mode
SIX : Shared and Intention Locks/Mode
```

SHARED LOCK: [S]



INTENTION SHARED MODE: [IS]

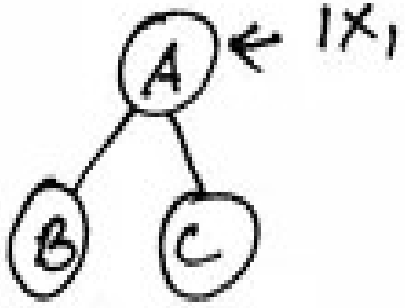


S: Shared Mode

IS: Intension shared Mode.

ie there is some node (say P_3) below F_1 (IS Mode, which is locked in shared mode.

INTENTION EXCLUSIVE MODE: [IX]



T_2

IX(A)

W(B) ← Denied

R(C) ← Denied

T_3

IX(A)

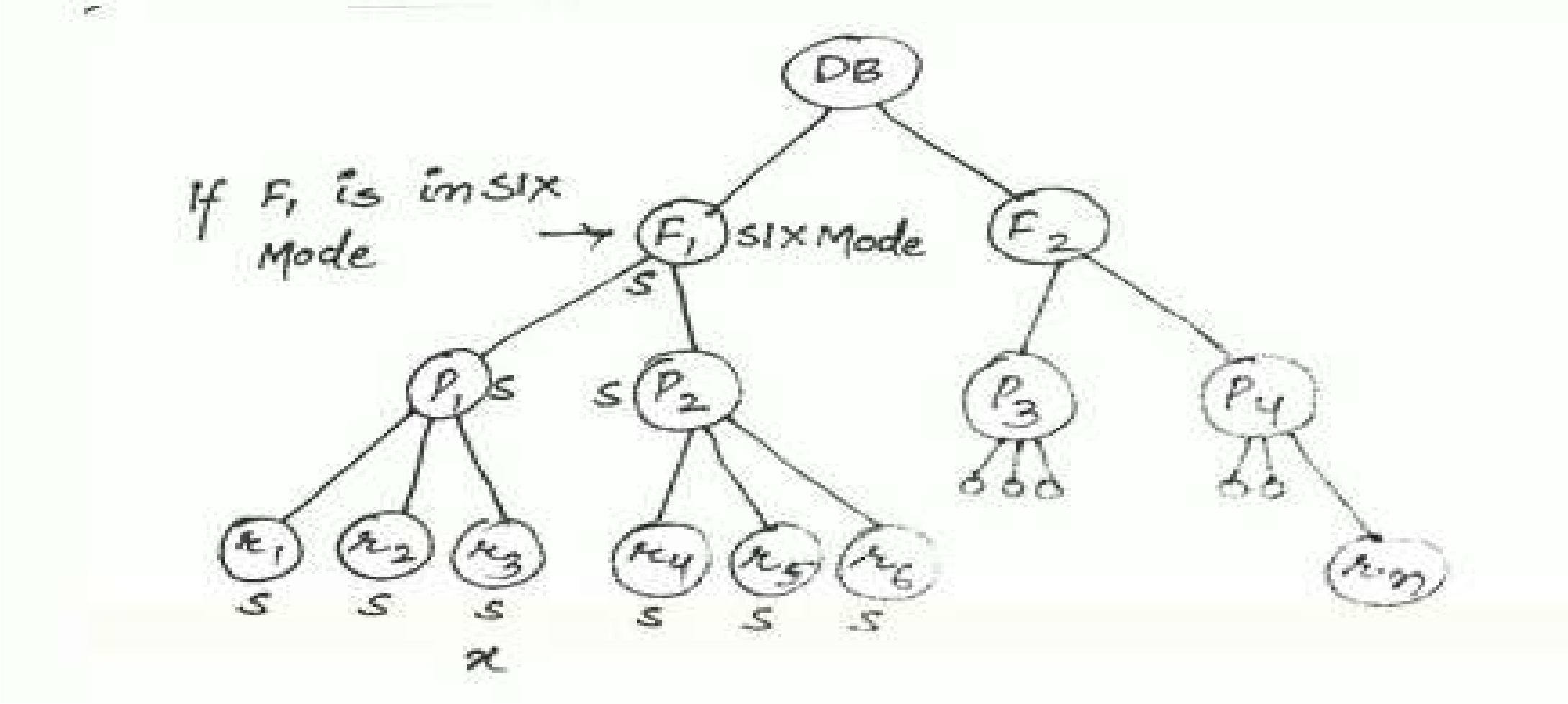
X(B)

W(B) ← Now write is granted

S(C)

R(C) ← Now write is granted.

SHARED AND INTENTION EXCLUSIVE MODE: [SIX]



THE LOCK COMPATIBILITY MATRIX