Module 5

Chapter 1: Concurrency Control in Databases

Chapter 1: Concurrency Control in Databases

5.0 Introduction to Concurrency Control

- Purpose of Concurrency Control
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.
- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

5.1 Two-Phase Locking Techniques for Concurrency Control

- The concept of locking data items is one of the main techniques used for controlling the concurrent execution of transactions.
- A lock is a variable associated with a data item in the database. Generally there is a lock for each data item in the database.
- A lock describes the status of the data item with respect to possible operations that can be applied to that item.
- It is used for synchronizing the access by concurrent transactions to the database items.
- A transaction locks an object before using it
- When an object is locked by another transaction, the requesting transaction must wait

5.1.1 Types of Locks and System Lock Tables

1. **Binary Locks**

- A **binary lock** can have two **states** or **values:** locked and unlocked (or 1 and 0).
- If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item

- If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1
- We refer to the current value (or state) of the lock associated with item X as **lock**(**X**).
- Two operations**, lock_item** and **unlock_item**, are used with binary locking.
- A transaction requests access to an item *X* by first issuing a **lock_item(*X*)** operation
- If LOCK(*X*) = 1, the transaction is forced to wait.
- If LOCK(*X*) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item *X*
- When the transaction is through using the item, it issues an **unlock_item(*X*)** operation, which sets LOCK(*X*) back to 0 (**unlocks** the item) so that *X* may be accessed by other transactions
- Hence, a binary lock enforces **mutual exclusion** on the data item.

```
lock_item(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);
             go to B
    end;
unlock_item(X):
LOCK(X) ← 0; (* unlock the item *)
if any transactions are waiting
then wakeup one of the waiting transactions ;
```

Fig: 2.1.1 Lock and unlock operations for binary licks.

- The lock_item and unlock_item operations must be implemented as indivisible units that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits
- The wait command within the lock_item(*X*) operation is usually implemented by putting the transaction in a waiting queue for item *X* until *X* is unlocked and the transaction can be granted access to it
- Other transactions that also want to access *X* are placed in the same queue.Hence, the wait command is considered to be outside the lock_item operation.
- It is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item X in the database
- In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item
- If the simple binary locking scheme described here is used, every transaction must obey the following rules:
    1. A transaction *T* must issue the operation lock_item(*X*) before any read_item(*X*) or write_item(*X*) operations are performed in *T.*
    2. A transaction *T* must issue the operation unlock_item(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T.*
    3. A transaction *T* will not issue a lock_item(*X*) operation if it already holds the lock on item *X*.
    4. A transaction *T* will not issue an unlock_item(*X*) operation unless it already holds the lock on item *X*.

2. Shared/Exclusive (or Read/Write) Locks

- binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item
- should allow several transactions to access the same item X if they all access X for reading purposes only
- if a transaction is to write an item *X*, it must have exclusive access to *X*
- For this purpose, a different type of lock called a **multiple-mode lock** is used
- In this scheme—called **shared/exclusive** or **read/write** locks —there are three locking operations: **read_lock(*X*), write_lock(*X*), and unlock(*X*)**.

- A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item
- Method to implement read/write lock is to
  - keep track of the number of transactions that hold a shared (read) lock on an item in the lock table
  - Each record in the lock table will have four fields:
    <Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>.
- If LOCK(*X*)=write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on *X*
- If LOCK(*X*)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on *X*.

```
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 LOCK(X) ← "write-locked"
    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";
                wakeup one of the waiting transactions, if any
                end
    else it 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";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

- When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.

3 A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.3

4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks

- A transaction that already holds a lock on item *X* is allowed under certain conditions to **convert** the lock from one locked state to another
- For example, it is possible for a transaction *T* to issue a read_lock(*X*) and then later to **upgrade** the lock by issuing a write_lock(*X*) operation

  - If *T* is the only transaction holding a read lock on *X* at the time it issues the write_lock(*X*) operation, the lock can be upgraded;otherwise, the transaction must wait

5.1.2 Guaranteeing Serializability by Two-Phase Locking

- A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction
- Such a transaction can be divided into two phases:
  - **Expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released
  - **Shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired
- If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.
- Transactions *T*1 and *T*2 in Figure 22.3(a) do not follow the two-phase locking protocol because the write_lock(*X*) operation follows the unlock(*Y*) operation in *T*1, and similarly the write_lock(*Y*) operation follows the unlock(*X*) operation in *T*2.

(a)

T_1	T_2

read_lock(*Y*);
read_item(*Y*);
unlock(*Y*);
write_lock(*X*);
read_item(*X*);
X := *X* + *Y*;
write_item(*X*);
unlock(*X*);
	read_lock(*X*);
	read_item(*X*);
	unlock(*X*);
	write_lock(*Y*);
	read_item(*Y*);
	Y := *X* + *Y*;
	write_item(*Y*);
	unlock(*Y*);

(b) Initial values: *X*=20, *Y*=30

Result serial schedule T_1 followed by T_2: *X*=50, *Y*=80

Result of serial schedule T_2 followed by T_1: *X*=70, *Y*=50

(c)

T_1	T_2

read_lock(*Y*);
read_item(*Y*);
unlock(*Y*);	

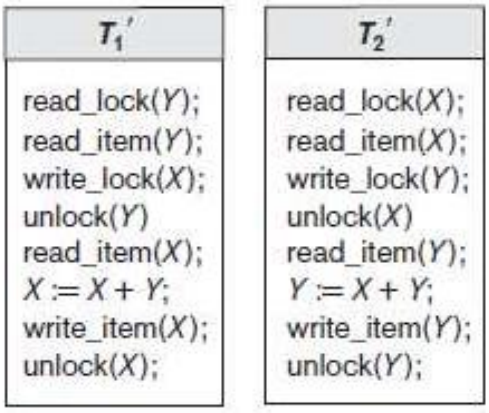
	read_lock(*X*);
	read_item(*X*);
	unlock(*X*);
	write_lock(*Y*);
	read_item(*Y*);
	Y := *X* + *Y*;
	write_item(*Y*);
	unlock(*Y*);

write_lock(*X*);
read_item(*X*);
X := *X* + *Y*;
write_item(*X*);
unlock(*X*);	

Time

Figure 21.3 Transactions that do not obey two-phase locking (a) Two transactions *T*1 and *T*2 (b) Results of possible serial schedules of *T*1 and *T*2 (c) A nonserializable schedule *S* that uses locks

- If we enforce two-phase locking, the transactions can be rewritten as *T*1' and *T*2' as shown in Figure 22.4.
- Now, the schedule shown in Figure 22.3(c) is not permitted for *T*1_ and *T*2_ (with their modified order of locking and unlocking operations) under the rules of locking because *T*1_ will issue its write_lock(*X*) *before* it unlocks item *Y*; consequently, when *T*2_ issues its read_lock(*X*), it is forced to wait until *T*1_ releases the lock by issuing an unlock (*X*) in the schedule.



T_1'	T_2'
read_lock(*Y*);	read_lock(*X*);
read_item(*Y*);	read_item(*X*);
write_lock(*X*);	write_lock(*Y*);
unlock(*Y*)	unlock(*X*)
read_item(*X*);	read_item(*Y*);
X := *X* + *Y*;	*Y* := *X* + *Y*;
write_item(*X*);	write_item(*Y*);
unlock(*X*);	unlock(*Y*);

Figure 22.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

- If every transaction in a schedule follows the two-phase locking protocol, schedule guaranteed to be serializable
- Two-phase locking may limit the amount of concurrency that can occur in a schedule
- Some serializable schedules will be prohibited by two-phase locking protocol

5.2 Concurrency Control Based on Timestamp Ordering

- guarantees serializability using transaction timestamps to order transaction execution for an equivalent serial schedule

5.2.1 Timestamps

- **timestamp** is a unique identifier created by the DBMS to identify a transaction.
- Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*.

We will refer to the timestamp of transaction *T* as **TS(*T*)**.

Concurrency control techniques based on timestamp ordering do not use locks;hence, *deadlocks cannot occur*.

Timestamps can be generated in several ways.

One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, ... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time.

- Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

5.2.2 The Timestamp Ordering Algorithm

- The idea for this scheme is to order the transactions based on their timestamps.
- A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values. This is called **timestamp ordering (TO**).
- The algorithm must ensure that, for each item accessed by *conflicting Operations* in the schedule, the order in which the item is accessed does not violate the timestamp order.
- To do this, the algorithm associates with each database item *X* two timestamp (**TS**) values:
    1. **read_TS(*X*).** The **read timestamp** of item *X* is the largest timestamp among all the timestamps of transactions that have successfully read item *X*—that is, read_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has read *X* successfully.
    2. **write_TS(*X*).** The **write timestamp** of item *X* is the largest of all the timestamps of transactions that have successfully written item *X*—that is, write_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has written *X* successfully.

Basic Timestamp Ordering (TO).

- Whenever some transaction *T* tries to issue a read_item(*X*) or a write_item(*X*) operation, the **basic TO** algorithm compares the timestamp of *T* with read_TS(*X*) and write_TS(*X*) to ensure that the timestamp order of transaction execution is not violated.
- If this order is violated, then transaction *T* is aborted and resubmitted to the system as a

new transaction with a *new timestamp*.

- If *T* is aborted and rolled back, any transaction *T*1 that may have used a value written by *T* must also be rolled back.
- Similarly, any transaction *T*2 that may have used a value written by *T*1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable.
- An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict.
- **The basic TO algorithm :**
  - The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Whenever a transaction *T* issues a write_item(*X*) operation, the following is checked:
   a. If read_TS(*X*) > TS(*T*) or if write_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the operation. This should be done because some *younger* transaction with a timestamp greater than TS(*T*)—and hence *after* *T* in the timestamp ordering—has already read or written the value of item *X* before *T* had a chance to write *X*, thus violating the timestamp ordering.
   b. If the condition in part (a) does not occur, then execute the write_item(*X*) operation of *T* and set write_TS(*X*) to TS(*T*).
2. Whenever a transaction *T* issues a read_item(*X*) operation, the following is checked:
   a. If write_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the operation. This should be done because some younger transaction with timestamp greater than TS(*T*)—and hence *after* *T* in the timestamp ordering—has already written the value of item *X* before *T* had a chance to read *X*.
   b. If write_TS(*X*) ≤ TS(*T*), then execute the read_item(*X*) operation of *T* and set read_TS(*X*) to the *larger* of TS(*T*) and the current read_TS(*X*).

- Whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*

Strict Timestamp Ordering (TO)

- A variation of basic TO called **strict TO** ensures that the schedules are both **strict** (for easy recoverability) and (conflict) serializable.

- In this variation, a transaction *T* that issues a read_item(*X*) or write_item(*X*) such that TS(*T*) > write_TS(*X*) has its read or write operation *delayed* until the transaction *T*' that *wrote* the value of *X* (hence TS(*T*') = write_TS(*X*)) has committed or aborted.
- To implement this algorithm, it is necessary to simulate the locking of an item *X* that has been written by transaction *T* ' until *T* is either committed or aborted. This algorithm *does not cause deadlock*, since *T* waits for *T* ' only if TS(*T*) > TS(*T*_).

Thomas's Write Rule

- A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(*X*) operation as follows:
  1. If read_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the operation.
  2. If write_TS(*X*) > TS(*T*), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(*T*)—and hence after *T* in the timestamp ordering—has already written the value of *X*. Thus, we must ignore the write_item(*X*) operation of *T* because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).

If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(*X*) operation of *T* and set write_TS(*X*) to TS(*T*).

5.3 Multiversion Concurrency Control Techniques

- Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as **multiversion concurrency control**, because several versions (values) of an item are maintained
- When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible.
- The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability.When a transaction writes an item, it writes a *new version* and the old version(s) of the item are retained
- An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items

5.3.1 Multiversion Technique Based on Timestamp Ordering

- In this method, several versions *X*1, *X*2, ..., *Xk* of each data item *X* are maintained.
- For *each version*, the value of version *Xi* and the following two timestamps are kept:
  1. **read_TS(*Xi*).** The **read timestamp** of *Xi* is the largest of all the timestamps of transactions that have successfully read version *Xi*.
  2. **write_TS(*Xi*).** The **write timestamp** of *Xi* is the timestamp of the transaction that wrote the value of version *Xi*.
- Whenever a transaction *T* is allowed to execute a write_item(*X*) operation, a new version *Xk*+1 of item *X* is created, with both the write_TS(*Xk*+1) and the read_TS(*Xk*+1) set to TS(*T*)
- Correspondingly, when a transaction *T* is allowed to read the value of version *Xi*, the value of read_TS(*Xi*) is set to the larger of the current read_TS(*Xi*) and TS(*T*).
- To ensure serializability, the following rules are used:
  1. If transaction *T* issues a write_item(*X*) operation, and version *i* of *X* has the highest write_TS(*Xi*) of all versions of *X* that is also *less than or equal to* TS(*T*), and read_TS(*Xi*) > TS(*T*), then abort and roll back transaction *T*; otherwise, create a new version *Xj* of *X* with read_TS(*Xj*) = write_TS(*Xj*) = TS(*T*).
  2. If transaction *T* issues a read_item(*X*) operation, find the version *i* of *X* that has the highest write_TS(*Xi*) of all versions of *X* that is also *less than or equal to* TS(*T*); then return the value of *Xi* to transaction *T,* and set the value of read_TS(*Xi*) to the larger of TS(*T*) and the current read_TS(*Xi*).

5.3.2 Multiversion Two-Phase Locking Using Certify Locks

- In this multiple-mode locking scheme, there are *three locking modes* for an item: read, write, and *certify*
- Hence, the state of LOCK(*X*) for an item *X* can be one of read-locked, writelocked, certify-locked, or unlocked
- We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 22.6(a)
- An entry of *Yes* means that if a transaction *T* holds the type of lock specified in the column header on item *X* and if transaction *T*_ requests the type of lock specified i
- the row header on the same item *X*, then *T*_ *can obtain the lock* because the locking modes are compatible

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 22.6: Lock compatibility tables. (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme.

- On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so *T' must wait* until *T releases* the lock
- The idea behind multiversion 2PL is to allow other transactions *T'* to read an item *X* while a single transaction *T* holds a write lock on *X*
- This is accomplished by allowing *two versions* for each item *X*; one version must always have been written by some committed transaction
- The second version *X'* is created when a transaction *T* acquires a write lock on the item

5.4 Validation (Optimistic) Concurrency Control Techniques

- In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing
- In this scheme, updates in the transaction are *not* applieddirectly to the database items until the transaction reaches its end
- During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction
- At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability.
- There are three phases for this concurrency control protocol:

1. **Read phase.** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase.** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase.** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

- The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached
- The techniques are called *optimistic* because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.
- The validation phase for *Ti* checks that, for *each* such transaction *Tj* that is either committed or is in its validation phase, *one* of the following conditions holds:
    1. Transaction *Tj* completes its write phase before *Ti* starts its read phase.
    2. *Ti* starts its write phase after *Tj* completes its write phase, and the read_set of *Ti* has no items in common with the write_set of *Tj*.
    3. Both the read_set and write_set of *Ti* have no items in common with the write_set of *Tj*, and *Tj* completes its read phase before *Ti* completes its read phase.

5.5 Granularity of Data Items and Multiple Granularity Locking

- All concurrency control techniques assume that the database is formed of a number of named data items. A database item could be chosen to be one of the following:
    - A database record
    - A field value of a database record
    - A disk block
    - A whole file
    - The whole database

- The granularity can affect the performance of concurrency control and recovery

5.5.1 Granularity Level Considerations for Locking

- The size of data items is often called the **data item granularity**.
- *Fine granularity* refers to small item sizes, whereas *coarse granularity* refers to large

item sizes

- The larger the data item size is, the lower the degree of concurrency permitted.
- For example, if the data item size is a disk block, a transaction *T* that needs to lock a record *B* must lock the whole disk block *X* that contains *B* because a lock is associated with the whole data item (block). Now, if another transaction *S* wants to lock a different record *C* that happens to reside in the same block *X* in a conflicting lock mode, it is forced to wait. If the data item size was a single record, transaction *S* would be able to proceed, because it would be locking a different data item (record).
- The smaller the data item size is, the more the number of items in the database. Because every item is associated with a lock, the system will have a larger number of active locks to be handled by the lock manager. More lock and unlock operations will be performed, causing a higher overhead
- The best item size *depends on the types of transactions involved*.
- If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record
- On the other hand, if a transaction typically accesses many records in the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items

5.5.2 Multiple Granularity Level Locking

Since the best granularity size depends on the given transaction, it seems appropriate that a database system should support multiple levels of granularity, where the granularity level can be different for various mixes of transactions

Figure 22.7 shows a simple granularity hierarchy with a database containing two files, each file containing several disk pages, and each page containing several records.

This can be used to illustrate a **multiple granularity level** 2PL protocol, where a lock can be requested at any level

-
-
-

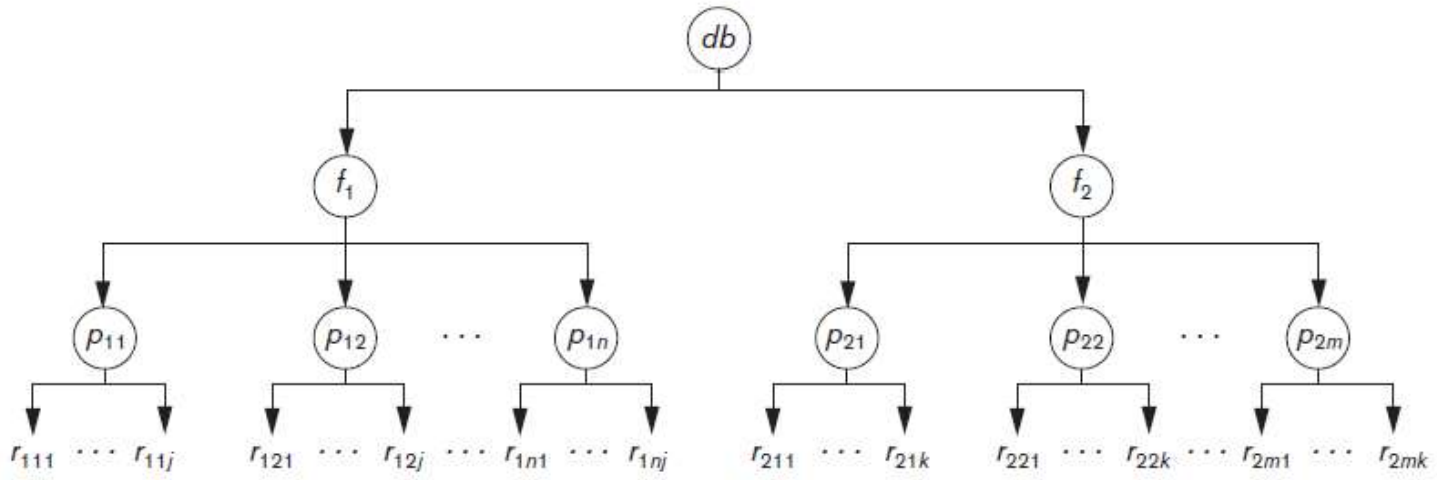


Figure 22.7 A granularity hierarchy for illustrating multiple granularity level locking

- To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed
- The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants.
- There are three types of intention locks:
  1. Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).
  2. Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).
  3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).
- The compatibility table of the three intention locks, and the shared and exclusive locks, is shown in Figure 22.8.

IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 22.8: Lock compatibility matrix for multiple granularity locking.

- The **multiple granularity locking (MGL)** protocol consists of the following rules:
  1. The lock compatibility (based on Figure 22.8) must be adhered to.
  2. The root of the tree must be locked first, in any mode.
  3. A node *N* can be locked by a transaction *T* in S or IS mode only if the parent node *N* is already locked by transaction *T* in either IS or IX mode.
  4. A node *N* can be locked by a transaction *T* in X, IX, or SIX mode only if the parent of node *N* is already locked by transaction *T* in either IX or SIX mode.
  5. A transaction *T* can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
  6. A transaction *T* can unlock a node, *N*, only if none of the children of node *N* are currently locked by *T.*
- The multiple granularity level protocol is especially suited when processing a mix of transactions that include

  (1) short transactions that access only a few items (records or fields) and

  (2) long transactions that access entire files.