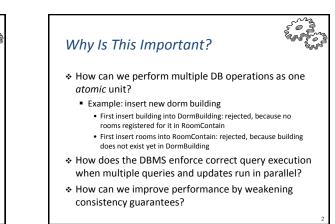


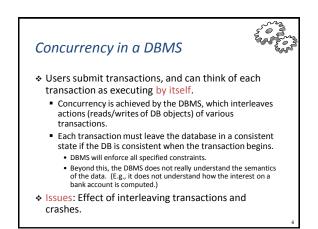
Chapter 16



### Transactions



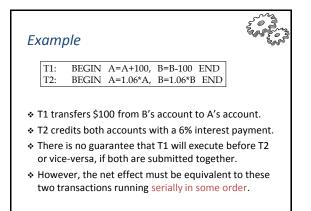
- Concurrent execution of user programs is essential for good DBMS performance.
  - While some request is waiting for I/O, CPU can work on another one.
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes.

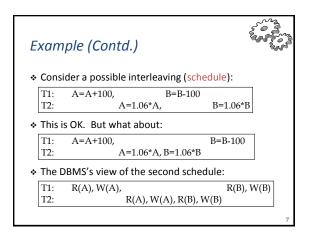


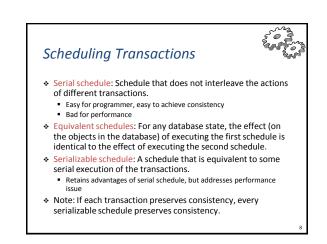
## The ACID Properties

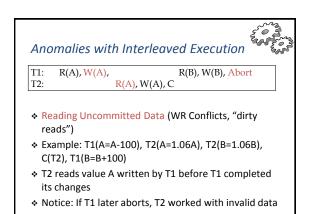


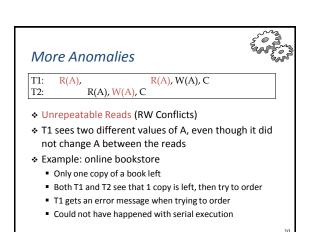
- Atomicity: Either all or none of the transaction's actions are executed
  - Even when a crash occurs mid-way
- Consistency: Transaction run by itself must preserve consistency of the database
  - User's responsibility
- Isolation: Transaction semantics do not depend on other concurrently executed transactions
- Durability: Effects of successfully committed transactions should persist, even when crashes occur

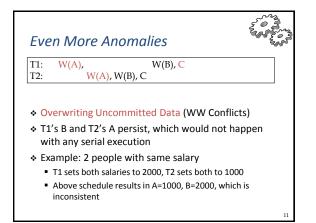


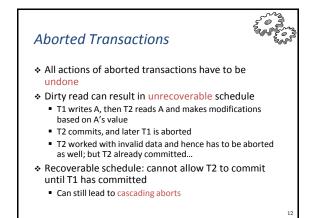












### Preventing Anomalies through Locking

- \* DBMS can support concurrent transactions while preventing anomalies by using a locking protocol
- If a transaction wants to read an object, it first requests a shared lock (S-lock) on the object
- If a transaction wants to modify an object, it first requests an exclusive lock (X-lock) on the object
- \* Multiple transactions can hold a shared lock on an object
- \* At most one transaction can hold an exclusive lock on an object

# Lock-Based Concurrency Control

- Strict Two-phase Locking (Strict 2PL) Protocol:
  - Each Xact must obtain the appropriate lock before accessing an object.
  - All locks held by a transaction are released when the transaction is completed.
  - All this happens automatically inside the DBMS
- Strict 2PL allows only serializable schedules.
  - Prevents all the anomalies shown earlier

### The Phantom Problem



- \* Assume initially the youngest sailor is 20 years old
- T1 contains this query twice
- SELECT rating, MIN(age) FROM Sailors
- T2 inserts a new sailor with age 18
- Consider the following schedule:
  - T1 runs query, T2 inserts new sailor, T1 runs query again T1 sees two different results! Unrepeatable read.
- Would Strict 2PL prevent this?
  - Assume T1 acquires Shared lock on each existing sailor tuple
  - T2 inserts a new tuple, which is not locked by T1
  - T2 releases its Exclusive lock on the new sailor before T1 reads Sailors again
- What went wrong?

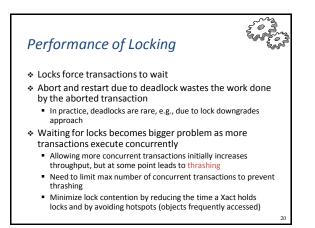
# What Should We Lock? T1 cannot lock a tuple that T2 will insert

- ...but T1 could lock the entire Sailors table
- Now T2 cannot insert anything until T1 completed
- What if T1 computed a slightly different query: SELECT MIN(age) FROM Sailors WHERE rating = 8
- Now locking the entire Sailors table seems excessive. because inserting a new sailor with rating <> 8 would not create a problem
  - T1 can lock the predicate [rating = 8] on Sailors
- General challenge: DBSM needs to choose appropriate granularity for locking

### Deadlocks



- Assume T1 and T2 both want to read and write objects A and B
  - T1 acquires X-lock on A; T2 acquires X-lock on B
  - Now T1 wants to update B, but has to wait for T2 to release its lock on B
  - But T2 wants to read A and also waits for T1 to release its lock on A
  - Strict 2PL does not allow either to release its locks before the transaction completed. Deadlock!
- DBMS can detect this
  - Automatically breaks deadlock by aborting one of the involved transactions
  - Tricky to choose which one to abort: work performed is lost



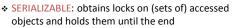
# Controlling Locking Overhead



- Declaring Xact as "READ ONLY" increases concurrency
- Isolation level: trade off concurrency against exposure of Xact to other Xact's uncommitted changes

Isolation Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

## Locking vs. Isolation Levels



- REPEATABLE READ: same locks as for serializable Xact, but does not lock sets of objects at higher level
- READ COMMITTED: obtains X-locks before writing and holds them until the end; obtains S-locks before reading, but releases them immediately after reading
- READ UNCOMMITTED: does not obtain S-locks for reading; not allowed to perform any writes
  - Does not request any locks ever

#### Summary



- Concurrency control is one of the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and can schedule actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- DBMS automatically undoes the actions of aborted transactions.
  - Consistent state: Only the effects of committed Xacts seen.