Properties of schedules
Concurrency control schemes

- The DBMS must provide a mechanism that will ensure all possible schedules are:
  - serializable
  - recoverable, and preferably cascadeless
- Concurrency control protocols ensure these properties.
Lock-Based Concurrency Control

- Lock - associated with some object
  - shared or exclusive
- Locking protocol - set of rules to be followed by each transaction to ensure good properties.
Lock Compatibility Matrix

Locks on a data item are granted based on a lock compatibility matrix:

<table>
<thead>
<tr>
<th>Request mode</th>
<th>Mode of Data Item</th>
<th>None</th>
<th>Shared</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Exclusive</td>
<td></td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

When a transaction requests a lock, it must wait (block) until the lock is granted.
## Transaction performing locking

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lock-X(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>lock-S(B)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>unlock(B)</td>
</tr>
</tbody>
</table>
Two-Phase Locking (2PL)

- Two-Phase Locking Protocol
  - Each Xact must obtain a \( S \) (shared) lock on object before reading, and an \( X \) (exclusive) lock on object before writing.
  - A transaction cannot request additional locks once it releases any locks.
    - growing phase
    - shrinking phase
Strict Two-Phase Locking (Strict 2PL)

- **Strict Two-phase Locking Protocol:**
  - Each Xact must obtain a *S* (shared) lock on object before reading, and an *X* (exclusive) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - All X (exclusive) locks acquired by a transaction must be held until completion (commit/abort).
Not admissible under 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>
Lock-based protocols

- 2PL ensures conflict serializability
  - Transactions can be ordered by their end of growing phase (called lock point)
  - A 2PL schedule is equivalent to the serial schedule where transactions ordered by lock point order.

- Strict 2PL ensures conflict serializable and cascadeless schedules
  - Writers hold an X lock until they commit.
### Schedule following strict 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(A)</td>
<td>S(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>X(C)</td>
<td>X(B)</td>
</tr>
<tr>
<td>R(C)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(C)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
Lock Management

- Lock and unlock requests are handled by the lock manager.
- Lock table entry (for an object):
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations.
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock.
Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Tend to be rare in practice.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection
Deadlock

- Deadlock must be prevented or avoided.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A)</td>
<td></td>
</tr>
<tr>
<td>X(B)</td>
<td>X(B)</td>
</tr>
<tr>
<td></td>
<td>granted</td>
</tr>
<tr>
<td></td>
<td>granted</td>
</tr>
<tr>
<td></td>
<td>queued</td>
</tr>
<tr>
<td></td>
<td>queued</td>
</tr>
</tbody>
</table>
Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
  - add edge when queueing a lock request,
  - remove edge when granting lock request.

- Periodically check for cycles in the waits-for graph

- Resolve by aborting a transaction on cycle, releasing its locks.
Deadlock Detection (Continued)

T1  T2  T3  T4
S(A) R(A) X(B) W(B)
S(B)   X(C) S(C) R(C)
       X(C) X(A) X(B)
Deadlock Prevention

- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - **Wait-Die**: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - **Wound-wait**: If Ti has higher priority, Tj aborts; otherwise Ti waits

- If a transaction re-starts, make sure it has its original timestamp (to avoid starvation of a transaction).
Performance of Locking

- Lock-based schemes resolve conflicting schedules by **blocking** and **aborting**
  - in practice few deadlocks and relatively few aborts
  - most of penalty from blocking
- To increase throughput
  - lock smallest objects possible
  - reduce time locks are held
  - reduce hotspots
Anomalies with Interleaved Execution

- Not all interleavings of operations are okay.

- Anomaly: two consistency-preserving committed transactions that lead to an inconsistent state.

- Types of anomalies:
  - Reading Uncommitted Data (WR Conflicts) “dirty reads”
  - Unrepeateable Reads (RW Conflicts)
  - Overwriting Uncommitted Data (WW Conflicts)
**Reading Uncommitted Data**

"Dirty Read"

Inconsistent result of A is exposed to transaction T2

<table>
<thead>
<tr>
<th>T1: Transfer</th>
<th>T2: Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>

56
Unrepeatable Reads

T1 could see two values for A, although it has not changed A itself.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>
Overwriting Uncommitted Data

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(B)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(B)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>
Transaction support in SQL

- Transaction automatically started for SELECT, UPDATE, CREATE
- Transaction ends with COMMIT or ROLLBACK (abort)
- SQL 99 supports SAVEPOINTS which are simple nested transactions
What should we lock?

- **Coarse locking:**
  - T1 S-lock on Sailors;
  - T2 X-lock on Sailors

- **Finer-grained locking:**
  - T1 S-lock on all rows with rating=8;
  - T2 X-lock on Joe’s tuple.

```sql
SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.rating = 8

UPDATE Sailors(Name, Rating, Age) VALUES ("Joe", 8, 33)
```
The Phantom Problem

Suppose T1 locks all existing rows with rating=8.
Then T3 creates row and sets X-lock.
These locks don’t conflict; T1 could read different results even though it did not modify them itself.
The Phantom Problem

- T1 locks all *existing* rows with rating=8.
- But a new row satisfying condition could be inserted
- Phantom problem: A transaction retrieves a collection of tuples and sees different results, even though it did not modify the tuples itself.
  - Conceptually: must lock all *possible* rows.
  - Can lock entire table.
  - Better, use index locking.
Index Locking

- If there is an index on the rating field using Alternative (2), T1 should lock the index page containing the data entries with rating = 8.
  - If there are no records with rating = 8, T1 must lock the index page where such a data entry would be, if it existed!

- If there is no suitable index, T1 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no new records with rating = 8 are added.
Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. \( \text{age} > 2 \times \text{salary} \).
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
- In general, predicate locking has a lot of locking overhead.
Specify isolation level

- General rules of thumb w.r.t. isolation:
  - Fully serializable isolation is more expensive than “no isolation”
    - We can’t do as many things concurrently (or we have to undo them frequently)
  - For performance, we generally want to specify the most relaxed isolation level that’s acceptable
    - Note that we’re “slightly” violating a correctness constraint to get performance!
Specifying isolation level in SQL

SET TRANSACTION [READ WRITE | READ ONLY] ISOLATION LEVEL [LEVEL];

LEVEL = SERIALIZABLE
       REPEATABLE READ
       READ COMMITTED
       READ UNCOMMITTED

Less isolation

The default isolation level is SERIALIZABLE

Locks sets of objects, avoids phantoms
REPEATABLE READ

- T reads only changes made by committed transactions
- No value read/written by T is changed by another transaction until T completes.

- Phantoms possible: inserts of qualifying tuples not avoided.

Locks only individual objects
READ COMMITTED

- T reads only changes made by committed transactions
- No value read/written by T is changed by another transaction until T completes.
- Value read by T may be modified while T is in progress.
- Phantoms possible.

X locks on written objects, held to end
S locks on read objects, but released immediately.
READ UNCOMMITTED

- Greatest exposure to other transactions
- Dirty reads possible
- Can’t make changes: must be READ ONLY
- Does not obtain shared locks before reading
  - Thus no locks ever requested.
# Summary of Isolation Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UN-COMMITTED</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System guarantees nice properties: ACID
  - This is implemented using a locking protocol
- Users can trade isolation for performance using SQL commands