# *Scheduling Transactions*

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

# *Serializable Schedule*

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| R(B) | |
| W(B) | |
| | R(B) |
| | W(B) |
| | Commit |
| Commit | |

# *When can actions be re-ordered?*

- Let I, J be two consecutive actions of T1 and T2
  - I=Read(O), J=Read(O)
  - I=Read(O), J=Write(O)
  - I=Write(O), J=Read(O)
  - I=Write(O), J=Write(O)
- If I and J are both reads, then they can be freely reordered.
- In all other cases, order impacts outcome of schedule.

# *Conflicting operations*

- Two operations **conflict** if:
  - they operate on the same data object, and
  - at least one is a WRITE.
- Schedule outcome is determined by order of the conflicting operations.

# *Conflict Serializable Schedules*

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions (of committed trans) are ordered the same way.
  - Alternatively: S can be transformed to S′ by swaps of non-conflicting actions.
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Every conflict serializable schedule is serializable.

(exception: dynamic databases)

# *Conflict-serializable schedule*

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| R(B) | |
| W(B) | |
| | R(B) |
| | W(B) |
| | Commit |
| Commit | |

# *Not conflict-serializable*

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| R(B) | |
| W(B) | |
| Commit | |

# *Precedence graphs*

- Directed graph derived from schedule S:
  - Vertex for each transaction
  - Edge from Ti to Tj if:
    - Ti executes Write(O) before Tj executes Read(O)
    - Ti executes Read(O) before Tj executes Write(O)
    - Ti executes Write(O) before Tj executes Write(O)

If edge Ti -> Tj appears in precedence graph, then in any serial schedule equivalent to S, Ti must appear before Tj.

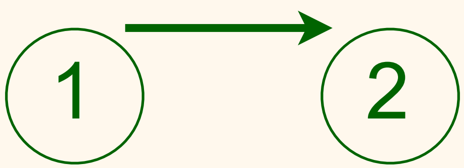
# *Dependency Graph*

- Theorem: A schedule is **conflict serializable** if and only if its dependency graph is acyclic.

(A serializable order can be found by topological sort of the dependency graph.)

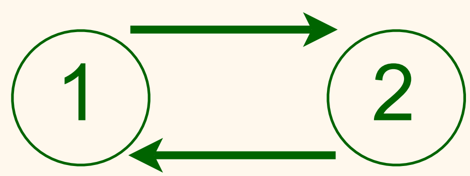
# *Construct precedence graphs:*

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| R(B) | |
| W(B) | |
| | R(B) |
| | W(B) |
| | Commit |
| Commit | |




Conflict serializable

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| R(B) | |
| W(B) | |
| Commit | |




Non-conflict serializable

# *Construct precedence graph:*

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | W(A) | |
| | Commit | |
| W(A) | | |
| Commit | | |
| | | W(A) |
| | | Commit |

# *Recoverable schedules*

- We must also consider the impact of transaction failures on concurrently running transactions.
  - That is, schedules with ABORT
- **Recoverable schedule**: if Tj reads data written by Ti, then *Ti commits before Tj commits*.

A Non-recoverable schedule.

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| Abort | |

DBMS must ensure recoverable schedules.

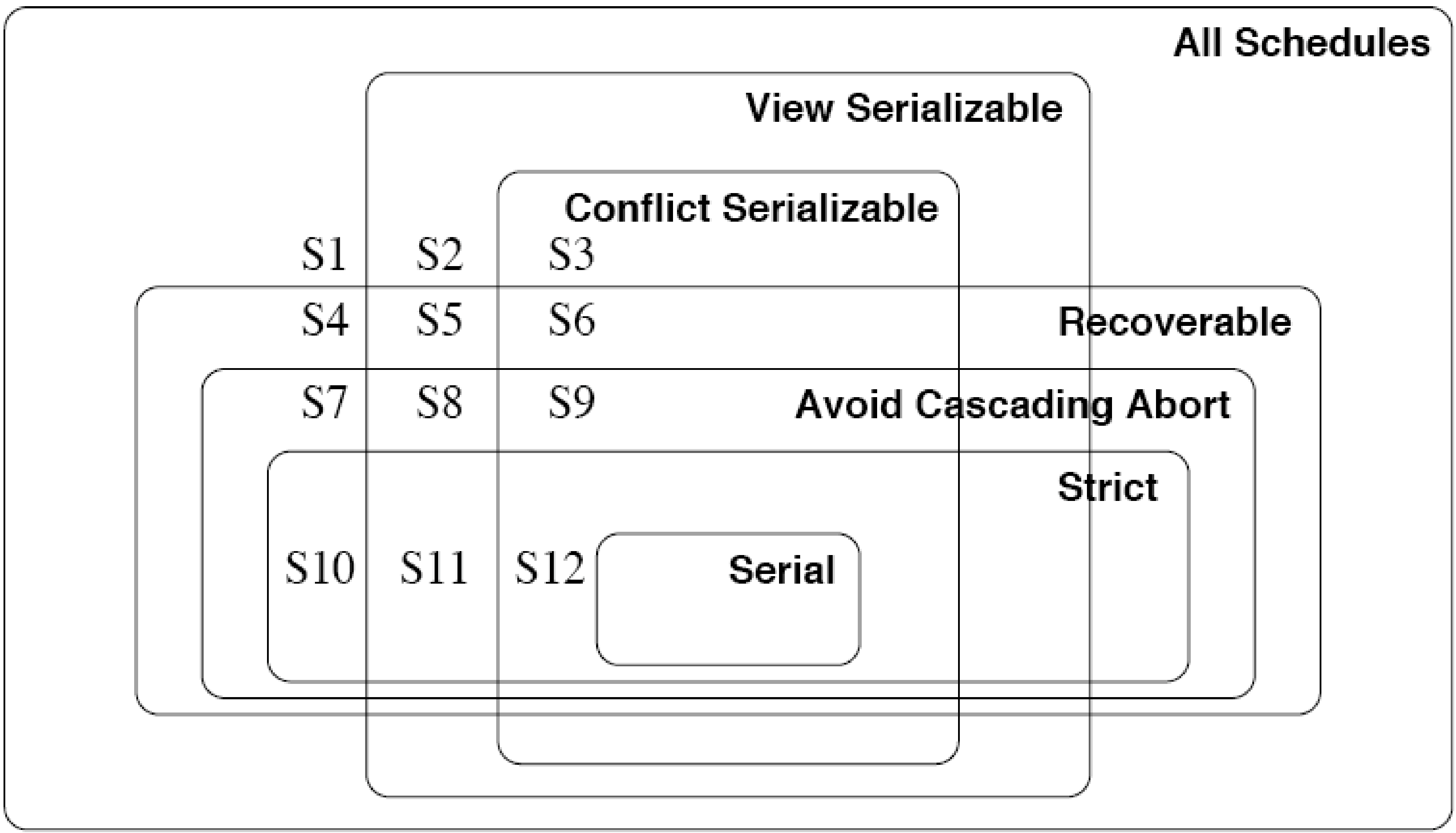
# *Cascadeless schedules*

- Even if schedule is recoverable, several transactions may need to be rolled back to recover correctly.
- **Cascading Rollback**: a single transaction failure leading to a series of rollbacks

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| R(B) | | |
| W(A) | | |
| | R(A) | |
| | W(A) | |
| | | R(A) |
| Abort | | |

- **Cascadeless schedule**: if Tj reads data written by Ti, then *Ti commits before read operation of Tj*.

# *Properties of schedules*



# *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"
  - Unrepeatable Reads (RW Conflicts)
  - Overwriting Uncommitted Data (WW Conflicts)

# *Reading Uncommitted Data*

"Dirty Read"

Inconsistent result of A is exposed to transaction T2

| T1: Transfer | T2: Interest |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| R(B) | |
| W(B) | |
| Commit | |

# *Acceptable Dirty Read*

If we are just checking availability of an airline seat, a dirty read might be fine! (*Why is that*?)

Reservation transaction:

```
EXEC SQL select occupied into :occ
            from Flights
            where Num= '123' and date=11-03-99
                  and seat='23f';
if (!occ) {EXEC SQL
           update Flights
           set occupied=true
           where Num= '123' and date=11-03-99
                 and seat='23f';}
else {notify user that seat is unavailable}
```

# *Unrepeatable Reads (RW Conflicts)*

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

| T1 | T2 |
|---|---|
| R(A) | |
| | R(A) |
| | W(A) |
| | Commit |
| R(A) | |
| W(A) | |
| Commit | |

# *Overwriting Uncommitted Data*

| T1 | T2 |
|---|---|
| | W(A) |
| W(B) | |
| | W(B) |
| | Commit |
| W(A) | |
| Commit | |

# *Schedules involving abort*

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| Abort | |

This is an unrecoverable schedule

- Recoverable schedule: transactions commit only after all transactions whose changes they <u>read</u> commit.

# *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:

| | | Mode of Data Item | | |
|---|---|---|---|---|
| | | None | Shared | Exclusive |
| Request mode | Shared | Y | Y | N |
| | Exclusive | Y | N | N |

When a transaction requests a lock, it must wait (block) until the lock is granted

# *Strict Two Phase Locking*

- *(Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading,
  - Each Xact must obtain an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Allows only safe interleavings
  - No anomalies

# *Schedule following strict 2PL*

| T1 | T2 |
|---|---|
| S(A) | |
| R(A) | |
| | S(A) |
| | R(A) |
| | X(B) |
| | R(B) |
| | W(B) |
| | Commit |
| X(C) | |
| R(C) | |
| W(C) | |
| Commit | |

# *Deadlock*

| T1 | T2 | |
|---|---|---|
| X(A) | | **granted** |
| | X(B) | **granted** |
| X(B) | | **queued** |
| | X(A) | **queued** |

- Deadlock must be prevented or avoided.

# *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

# *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?*

**T1**

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

**T2**

UPDATE Sailors(Name, Rating, Age) VALUES ("Joe", 8, 33)

- T1 **S**-lock on Sailors; T2 **X**-lock on Sailors
- T1 **S**-lock on all rows with rating=8; T2 X-lock on Joe's tuple.

# *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.

# *The Phantom Problem*

**T1**

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

**T2**

```
UPDATE Sailors(Name, Rating,
Age) VALUES (“Joe”, 8, 33)
```

**T3**

```
INSERT Sailors(Name, Rating,
Age) VALUES (“Mary”, 8, 18)
```

- Suppose T1 locks all **existing** rows with rating=8.
- Then T3 creates row and sets X-lock
- T1 returns an answer that depends on its order relative to T3, but locking does not impose a relative order on these transactions.

# *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 UNCOMMITED

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 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 every requested.

## *Specifying Acceptable Isolation Levels*

- To signal to the system that a dirty read is acceptable,

```
SET TRANSACTION READ WRITE
ISOLATION LEVEL READ UNCOMMITTED;
```

- In addition, there are

-
```
SET TRANSACTION
ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION
ISOLATION LEVEL REPEATABLE READ;
```

# *Summary of Isolation Levels*

| Level | Dirty Read | Unrepeatable Read | Phantoms |
|---|---|---|---|
| READ UN-COMMITTED | Maybe | Maybe | Maybe |
| READ COMMITTED | No | Maybe | Maybe |
| REPEATABLE READ | No | No | Maybe |
| SERIALIZABLE | No | No | No |

## *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