Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order $n^2$ time, where $n$ is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where $e$ is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?
Precedence Graphs and Conflict Equivalence: Lemma

• S1, S2 conflict equivalent ⇒ P(S1)=P(S2)

Proof sketch:
Assume P(S₁) ≠ P(S₂)
⇒ ∃ Ti: Ti → Tj in S₁ and not in S₂
⇒ S₁ = …pᵢ(A)... qᵢ(A)... pᵢ, qᵢ
S₂ = …qⱼ(A)...pᵢ(A)... conflict

⇒ S₁, S₂ not conflict equivalent

Note: P(S₁)=P(S₂) ↛ S₁, S₂ conflict equivalent

Counter example:

S₁=w₁(A) r₂(A) w₂(B) r₁(B)
S₂=r₂(A) w₁(A) r₁(B) w₂(B)
Theorem

- \( P(S_1) \) acyclic \( \iff \) \( S_1 \) conflict serializable

(\( \iff \)) Assume \( S_1 \) is conflict serializable

\( \Rightarrow \) \( \exists \ S_s: S_s, S_1 \) conflict equivalent

\( \Rightarrow P(S_s) = P(S_1) \)

\( \Rightarrow P(S_1) \) acyclic since \( P(S_s) \) is acyclic

Theorem:

\( P(S_1) \) acyclic \( \iff \) \( S_1 \) conflict serializable

(\( \Rightarrow \)) Assume \( P(S_1) \) is acyclic

Transform \( S_1 \) as follows:

1. Take \( T_1 \) to be transaction with no incident arcs
2. Move all \( T_1 \) actions to the front
   \( S_1 = \ldots q_1(A) \ldots \ldots p_1(A) \ldots \)
3. we now have \( S_1 = < T_1 \) actions > < rest ...
4. repeat above steps to serialize rest!
Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of $NP$-complete problems.
  - Thus, existence of an efficient algorithm is extremely unlikely.
- However practical algorithms that just check some sufficient conditions for view serializability can still be used.

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - What do we do if the schedule isn’t serializable?
- Testing a schedule for serializability after it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.
Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction $T_i$ reads a data item previously written by a transaction $T_j$, then the commit operation of $T_j$ appears before the commit operation of $T_i$.

- The following schedule (Schedule 11) is not recoverable

<table>
<thead>
<tr>
<th>$T_8$</th>
<th>$T_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ($A$)</td>
<td>read ($A$)</td>
</tr>
<tr>
<td>write ($A$)</td>
<td>commit</td>
</tr>
<tr>
<td>read ($B$)</td>
<td></td>
</tr>
</tbody>
</table>

- If $T_8$ should abort, $T_9$ would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

<table>
<thead>
<tr>
<th>$T_{10}$</th>
<th>$T_{11}$</th>
<th>$T_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ($A$)</td>
<td>read ($A$)</td>
<td>read ($A$)</td>
</tr>
<tr>
<td>read ($B$)</td>
<td>write ($A$)</td>
<td></td>
</tr>
<tr>
<td>write ($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If $T_{10}$ fails, $T_{11}$ and $T_{12}$ must also be rolled back.
- Can lead to the undoing of a significant amount of work
Concurrent Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

We’ll cover this next, but first, some final words on transactions

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance
Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)
Transaction Definition in SQL

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- `connection.setAutoCommit(false);`
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
  - E.g. In SQL `set transaction isolation level serializable`
  - E.g. in JDBC -- `connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)`

Transactions as SQL Statements

- E.g., Transaction 1:
  ```sql
  select ID, name from instructor where salary > 90000
  ```
- E.g., Transaction 2:
  ```sql
  insert into instructor values ('11111', 'James', 'Marketing', 100000)
  ```
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?
  - Instance of the **phantom phenomenon**
- Also consider T3 below, with Wu’s salary = 90000
  ```sql
  update instructor
  set salary = salary * 1.1
  where name = 'Wu'
  ```
- Key idea: Detect “**predicate**” conflicts, and use some form of “**predicate locking**”