Transactions and Concurrency Control
To open two SQLDevelopers:

**On the Mac** do the following:

- click on the SQLDeveloper icon to start one instance
- from the command line run the following to start the second instance

```
/Applications/SQLDeveloper\ 4.1.0.18.37.app/Contents/Resources/sqldeveloper/sqldeveloper/bin/sqldeveloper
```

or

- open -n -a SQLDeveloper  # This should work if you have only one instance of SQLDevleoper on your machine, otherwise, you might get the wrong version.

**On Windows**, I think your can just click on the SQLDeveloper icon twice.
Types of Transactions

- Read Committed
- Read Uncommitted – not allowed in Oracle
- Repeatable Read
- Serializable Transactions
- ACID Transactions
- Phantoms
- Deadlocks
Read Committed

Transaction 1

```sql
select empno, ename, sal
from emp
where empno = 7839;
```

Pause "Press Enter to continue"

```sql
select empno, ename, sal
from emp
where empno = 7839;
```

Run Transaction 1 up to the “pause”, then run Transaction 2, then hit “Enter” in Transaction 1

Transaction 2

```sql
update emp set sal = sal + 1
where empno = 7839;

select empno, ename, sal from emp
where empno = 7839;
commit;
```

- Read Committed
- Read Uncommitted – not allowed in Oracle
- Repeatable Read
- Serializable Transactions
- ACID Transactions
- Phantoms
- Deadlocks
Read Uncommitted – not allowed in Oracle

Transaction 1

select empno, ename, sal
from emp
where empno = 7839;
pause "Press Enter to continue";
select empno, ename, sal
from emp
where empno = 7839;

Transaction 2

update emp set sal = sal + 1
where empno = 7839;
select empno, ename, sal from emp
where empno = 7839;
--commit;

Run Transaction 1 up to the “pause”, then run Transaction 2, then hit “Enter” in Transaction 1
Repeatable Read

Transaction 1
commit;
set transaction read only;
select empno, ename, sal from emp
  where empno = 7839;
pause "Press Enter to continue";
select empno, ename, sal from emp
  where empno = 7839;
commit;

Transaction 2
update emp set sal = sal + 1
  where empno = 7839;
select empno, ename, sal from emp
  where empno = 7839;
commit;

Run Transaction 1 up to the “pause”, then run Transaction 2, then hit “Enter” in Transaction 1
Serializable Transactions

Transaction 1

commit;
set transaction isolation level serializable;
select empno, ename, sal from emp
  where empno = 7839;
update emp set sal = sal + 10
  where empno = 7839;
select empno, ename, sal from emp
  where empno = 7839;
pause "Press Enter to continue";
select empno, ename, sal from emp
  where empno = 7839;
commit;

Compare starting Transaction 1 before and after starting Transaction 2

Transaction 2

commit;
set transaction isolation level serializable;
select empno, ename, sal from emp
  where empno = 7839;
update emp set sal = sal + 1
  where empno = 7839;
select empno, ename, sal from emp
  where empno = 7839;
pause "Press Enter to continue";
select empno, ename, sal from emp
  where empno = 7839;
commit;
ACID Properties

To preserve integrity of data, the database system must ensure:

**Atomicity.** Either all operations of the transaction are properly done in the database or none are.

**Consistency.** See the example on the next page.

**Isolation.** Each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

**Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
commit;
set transaction isolation level serializable;
select empno, ename, sal from emp
  where empno = 7839;
update emp set sal = sal + 10
  where empno = 7839;
select empno, ename, sal from emp
  where empno = 7839;
pause "Press Enter to continue";
select empno, ename, sal from emp
  where empno = 7839;
commit;

set serverout on
select empno, ename, sal from emp
  where empno = 7499;
select empno, ename, sal from emp
  where empno = 7839;
begin
  commit;
set transaction isolation level serializable;
update emp set sal = sal + 1
  where empno = 7499;
update emp set sal = sal + 1
  where empno = 7839;
commit;
dbms_output.put_line('*** Commit done.');
exception
  when others
  then rollback;
  dbms_output.put_line('*** Rollback done.');
end;
/
select empno, ename, sal from emp
  where empno = 7499;
select empno, ename, sal from emp
  where empno = 7839;

• Read Committed
• Read Uncommitted – not allowed in Oracle
• Repeatable Read
• Serializable Transactions
• ACID Transactions
• Phantoms
• Deadlocks
ACID Transactions—not really

commit;
set transaction isolation level serializable;
select empno, ename, sal from emp
  where empno = 7839;
update emp set sal = sal + 10
  where empno = 7839;
select empno, ename, sal from emp
  where empno = 7839;
pause "Press Enter to continue";
select empno, ename, sal from emp
  where empno = 7839;
commit;

set serverout on
select empno, ename, sal from emp
  where empno = 7499;
select empno, ename, sal from emp
  where empno = 7839;
--begin
commit;
set transaction isolation level serializable;
update emp set sal = sal + 1
  where empno = 7499;
update emp set sal = sal + 1
  where empno = 7839;
commit;

--dbms_output.put_line('*** Commit done.');
--exception
--when others
--then rollback;
--dbms_output.put_line('*** Rollback done.');
--end;
--/
select empno, ename, sal from emp
  where empno = 7499;
select empno, ename, sal from emp
  where empno = 7839;
ACID Transactions

commit;
set transaction isolation level serializable;
select empno, ename, sal from emp
 where empno = 7839;
update emp set sal = sal + 10
 where empno = 7839;
select empno, ename, sal from emp
 where empno = 7839;
pause "Press Enter to continue";
select empno, ename, sal from emp
 where empno = 7839;
commit;

set serverout on
select empno, ename, sal from emp
 where empno = 7499;
select empno, ename, sal from emp
 where empno = 7839;
begin
commit;
set transaction isolation level serializable;
update emp set sal = sal + 1
 where empno = 7499;
update emp set sal = sal + 1
 where empno = 7839;
commit;
--dbms_output.put_line('*** Commit done.') ;
--exception
--when others
--then rollback ;
--dbms_output.put_line('*** Rollback done.') ;
end ;
/
select empno, ename, sal from emp
 where empno = 7499;
select empno, ename, sal from emp
 where empno = 7839;

• Read Committed
• Read Uncommitted – not allowed in Oracle
• Repeatable Read
• Serializable Transactions
• ACID Transactions
• Phantoms
• Deadlocks
Row Level Granularity

Transaction 1

commit;
set transaction isolation level serializable;
select empno, ename, sal from emp
  where empno = 7839;
update emp set sal = sal + 10
  where empno = 7839;
select empno, ename, sal from emp
  where empno = 7839;
pause "Press Enter to continue";
select empno, ename, sal from emp
  where empno = 7839;
commit;

Transaction 2

commit;
set transaction isolation level serializable;
select empno, ename, sal from emp
  where empno = 7934;
update emp set sal = sal + 1
  where empno = 7934;
select empno, ename, sal from emp
  where empno = 7934;
pause "Press Enter to continue";
select empno, ename, sal from emp
  where empno = 7934;
commit;
Update Phantom Demo

Transaction 1

```sql
select empno, ename, sal from emp where sal > 4000;
pause "Press Enter to continue";
select empno, ename, sal from emp where sal > 4000;
```

Transaction 2

```sql
update emp set sal = 5000 where empno = 7844;
commit;
```
Update Phantom Demo

Reset demo database

- Read Committed
- Read Uncommitted – not allowed in Oracle
- Repeatable Read
- Serializable Transactions
- ACID Transactions
- Phantoms
- Deadlocks
Update Phantom Demo

Transaction 1

commit;
set transaction read only;
select empno, ename, sal from emp

where sal > 4000;
pause "Press Enter to continue";
select empno, ename, sal from emp

where sal > 4000;

Transaction 2

update emp set sal = 5000

    where empno = 7844;
commit;

• Read Committed
• Read Uncommitted – not allowed in Oracle
• Repeatable Read
• Serializable Transactions
• ACID Transactions
• Phantoms
• Deadlocks
Dirty Write

The possibility that two transactions might both perform an update on the same row before either transaction terminates

**Table 8-4 Read Committed vs. Serializable Transaction**

<table>
<thead>
<tr>
<th></th>
<th>Read Committed</th>
<th>Serializable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty write</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
<tr>
<td>Dirty read</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
<tr>
<td>Non-repeatable read</td>
<td>Possible</td>
<td>Not Possible</td>
</tr>
<tr>
<td>Phantoms</td>
<td>Possible</td>
<td>Not Possible</td>
</tr>
<tr>
<td>Compliant with ANSI/ISO SQL 92</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Read snapshot time</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>Transaction set consistency</td>
<td>Statement level</td>
<td>Transaction level</td>
</tr>
<tr>
<td>Row-level locking</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Readers block writers</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Writers block readers</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Different-row writers block writers</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Same-row writers block writers</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Waits for blocking transaction</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Subject to &quot;can't serialize access&quot; error</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Error after blocking transaction aborts</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Error after blocking transaction commits</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
ACID Properties

To preserve integrity of data, the database system must ensure:

**Atomicity.** Either all operations of the transaction are properly done in the database or none are.

**Consistency.** See the example on the next page.

**Isolation.** Each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

**Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
Example of Fund Transfer Transaction

Transaction to transfer $50 from account \( A \) to account \( B \):
1. \( \text{read}(A) \)
2. \( A := A - 50 \)
3. \( \text{write}(A) \)
4. \( \text{read}(B) \)
5. \( B := B + 50 \)
6. \( \text{write}(B) \)

**Atomicity** requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not done in the database, else an inconsistency will result.

**Consistency** requirement – the sum of \( A \) and \( B \) is unchanged by the execution of the transaction.
Example of Fund Transfer Transaction

Transaction to transfer $50 from account $A$ to account $B$:  
1. \textit{read}(A) 
2. $A := A - 50$ 
3. \textit{write}(A) 
4. \textit{read}(B) 
5. $B := B + 50$ 
6. \textit{write}(B) 

\textbf{Isolation} requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it would see an inconsistent database (the sum $A + B$ will be less than it should be) and this would violate the notion of isolation. 

Isolation can be ensured trivially by running transactions \textit{serially}.
Example of Fund Transfer Transaction

Transaction to transfer $50 from account $A$ to account $B$:
1. $\text{read}(A)$
2. $A := A - 50$
3. $\text{write}(A)$
4. $\text{read}(B)$
5. $B := B + 50$
6. $\text{write}(B)$

**Durability** requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
Schedules

*Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed

- a schedule for a set of transactions must consist of all instructions of those transactions
- must preserve the order in which the instructions appear in each individual transaction.
Example Schedule

Let $T_1$ transfer $50$ from $A$ to $B$, and $T_2$ transfer 10% of the balance from $A$ to $B$. The following is a **serial schedule** (Schedule 1 in the text), in which $T_1$ is followed by $T_2$.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>$A = A - 50$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td>$B = B + 50$</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>

Let $T_1$ transfer $50$ from $A$ to $B$, and $T_2$ transfer 10% of the balance from $A$ to $B$. The following is a **serial schedule** (Schedule 1 in the text), in which $T_1$ is followed by $T_2$.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$A$</th>
<th>$B$</th>
<th>$A + B$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Start</td>
<td>Start</td>
<td>Start</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>200</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A = A - 50$</td>
<td>$A = A - t$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B = B + 50$</td>
<td>$B = B + t$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>temp := $A \times 0.1$</td>
<td>t = $A \times 0.1$</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>$A := A - temp$</td>
<td>$A = A - t$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
<td></td>
<td></td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B = B + t$</td>
<td>$B = B + t$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
<td></td>
<td></td>
<td></td>
<td>255</td>
</tr>
</tbody>
</table>

Finish:

- $T_1$: 45
- $T_2$: 255
- $A + B$: 300
Example Serializable Schedule

Let $T_1$ and $T_2$ be the transactions defined previously. The following schedule is not a serial schedule, but it is equivalent to Schedule 1.

\[
\begin{array}{|c|c|}
\hline
T_1 & T_2 \\
\hline
\text{read}(A) & \text{read}(A) \\
A := A - 50 & A = A - 50 \\
\text{write}(A) & \text{write}(A) \\
\text{read}(B) & \text{read}(A) \\
B := B + 50 & t = A \times 0.1 \\
\text{write}(B) & A = A - t \\
\text{read}(B) & \text{write}(A) \\
B := B + \text{temp} & 45 \\
\text{write}(B) & \text{write}(B) \\
\hline
\end{array}
\]

In both Schedules 1 and 3, the sum $A + B$ is preserved.
The following concurrent schedule does not preserve the value of the sum $A + B$.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>$A = A - 50$</td>
</tr>
<tr>
<td>write($A$)</td>
<td></td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>temp := $A \times 0.1$</td>
</tr>
<tr>
<td></td>
<td>$A := A - \text{temp}$</td>
</tr>
<tr>
<td></td>
<td>write($A$)</td>
</tr>
<tr>
<td></td>
<td>read($B$)</td>
</tr>
<tr>
<td></td>
<td>$B := B + 50$</td>
</tr>
<tr>
<td></td>
<td>write($B$)</td>
</tr>
<tr>
<td></td>
<td>$B := B + \text{temp}$</td>
</tr>
<tr>
<td></td>
<td>write($B$)</td>
</tr>
</tbody>
</table>
Serializability

Basic Assumption – Each transaction preserves database consistency. Thus serial execution of a set of transactions (i.e., serial schedule) preserves database consistency. A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. conflict serializability
2. view serializability (We won’t study this)

We ignore operations other than read and write instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only read and write instructions.
Conflicts

Instructions $l_i$ and $l_j$ of transactions $T_i$ and $T_j$ respectively, conflict if and only if there exists some item $Q$ accessed by both $l_i$ and $l_j$, and

1. $l_i = \text{read}(Q), \ l_j = \text{read}(Q)$. $l_i$ and $l_j$ don’t conflict.
2. $l_i = \text{read}(Q), \ l_j = \text{write}(Q)$. They conflict.
3. $l_i = \text{write}(Q), \ l_j = \text{read}(Q)$. They conflict
4. $l_i = \text{write}(Q), \ l_j = \text{write}(Q)$. They conflict

Intuitively, a conflict between $l_i$ and $l_j$ forces a (logical) temporal order between them. If $l_i$ and $l_j$ are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.
Testing for Serializability

Consider some schedule of a set of transactions $T_1, T_2, ..., T_n$

**Precedence graph** — a directed graph where the vertices are the transactions (names). Draw an arc from $T_i$ to $T_j$ if the two transactions conflict, and $T_i$ accessed the data item on which the conflict arose earlier.

We may label the arc by the item that was accessed.

**Example 1**

![Diagram of T1 and T2 with arcs labeled x and y]
Example Schedule (Schedule A)

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(X)</td>
<td>read(Y)</td>
<td>read(Z)</td>
<td>read(V)</td>
<td>read(W)</td>
<td>read(W)</td>
</tr>
<tr>
<td>read(Y)</td>
<td>write(Y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(U)</td>
<td>write(U)</td>
<td>write(Z)</td>
<td>read(Y)</td>
<td>write(Y)</td>
<td>write(Z)</td>
</tr>
</tbody>
</table>
Precedence Graph for Schedule A
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.)
Recoverability

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

The following schedule is not recoverable if $T_9$ commits immediately after the Write(A) because 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.

<table>
<thead>
<tr>
<th>T8</th>
<th>T9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td>Write(A)</td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
</tr>
</tbody>
</table>
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>write($A$)</td>
</tr>
<tr>
<td>write($A$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If $T_{10}$ fails, $T_{11}$ and $T_{12}$ must also be rolled back. This can lead to the undoing of a significant amount of work.
Cascadedless schedules — cascading rollbacks cannot occur; for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads a data item previously written by $T_i$, the commit operation of $T_i$ appears before the read operation of $T_j$.

Every cascadeless schedule is also recoverable.

It is desirable to restrict the schedules to those that are cascadeless.
Concurrency Control

- Schedules must be conflict serializable, and recoverable, for the sake of database consistency, 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.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated.
Concurrency Control

How is concurrency control actually implemented in a DBMS?

• **Locking**

• Timestamp

• Multiversion Two-Phase Locking
The Two-Phase Locking Protocol

This is a protocol which ensures conflict-serializable schedules.

Phase 1: Growing Phase
• transaction may obtain locks
• transaction may not release locks

Phase 2: Shrinking Phase
• transaction may release locks
• transaction may not obtain locks
Cascading rollback is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks until it commits/aborts. **Rigorous two-phase locking** is even stricter: here all locks are held until commit/abort.
There can be conflict serializable schedules that cannot be obtained if two-phase locking is used but its not bad.
Consider the partial schedule

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X($B$)</td>
<td>lock-S($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$B := B - 50$</td>
<td>lock-S($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
</tbody>
</table>

Neither $T_3$ nor $T_4$ can make progress — executing lock-S($B$) causes $T_4$ to wait for $T_3$ to release its lock on $B$, while executing lock-X($A$) causes $T_3$ to wait for $T_4$ to release its lock on $A$. Such a situation is called a **deadlock**. To handle a deadlock one of $T_3$ or $T_4$ must be rolled back and its locks released.
Deadlock Detection

Deadlocks can be described as a *wait-for graph*, which consists of a pair \( G = (V, E) \),

- \( V \) is a set of vertices (all the transactions in the system)
- \( E \) is a set of edges; each element is an ordered pair \( T_i \rightarrow T_j \).

If \( T_i \rightarrow T_j \) is in \( E \), then there is a directed edge from \( T_i \) to \( T_j \), implying that \( T_i \) is waiting for \( T_j \) to release a data item.

When \( T_i \) requests a data item currently being held by \( T_j \), then the edge \( T_i \rightarrow T_j \) is inserted in the wait-for graph. This edge is removed only when \( T_j \) is no longer holding a data item needed by \( T_i \).

The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.
Deadlock Detection (Cont.)

Wait-for graph without a cycle

Wait-for graph with a cycle
Dead Lock Demo

Transaction 1

commit;
set transaction isolation level serializable;
update emp set sal = sal + 10
    where empno = 7839;
pause "Press Enter to continue";
update emp set sal = sal + 10
    where empno = 7934;
select empno, ename, sal from emp
    where empno = 7839;
select empno, ename, sal from emp
    where empno = 7934;
commit;

Transaction 2

commit;
set transaction isolation level serializable;
update emp set sal = sal + 1
    where empno = 7934;
update emp set sal = sal + 1
    where empno = 7839;
select empno, ename, sal from emp
    where empno = 7934;
select empno, ename, sal from emp
    where empno = 7839;
pause "Press Enter to continue";
commit;
Concurrency Control

How is concurrency control actually implemented in a DBMS?

- Locking
- Timestamp
- Multiversion Two-Phase Locking
Correctness of Timestamp-Ordering

The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

\[ \text{transaction with smaller timestamp} \rightarrow \text{transaction with larger timestamp} \]

Thus, there will be no cycles in the precedence graph. Timestamp protocol ensures freedom from deadlock as no transaction ever waits. But the schedule may not be cascade-free, and may not even be recoverable.
Concurrency Control

How is concurrency control actually implemented in a DBMS?

- Locking
- Timestamp
- **Multiversion Two-Phase Locking**
Multiversion Schemes

Multiversion schemes keep old versions of data item to increase concurrency.

Multiversion Timestamp Ordering

Multiversion Two-Phase Locking

Each successful write results in the creation of a new version of the data item written. Use timestamps to label versions. When a read($Q$) operation is issued, select an appropriate version of $Q$ based on the timestamp of the transaction, and return the value of the selected version. 

reads never have to wait as an appropriate version is returned immediately.
Oracle Concurrency Control

Multi-version Two Phase Locking

Comes from the use of the Oracle Undo Facility
# Undo Data Versus Redo Data

<table>
<thead>
<tr>
<th></th>
<th>Undo</th>
<th>Redo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record of</td>
<td>How to undo a change</td>
<td>How to reproduce a change</td>
</tr>
<tr>
<td>Used for</td>
<td>Rollback, read consistency, flashback</td>
<td>Rolling forward database changes</td>
</tr>
<tr>
<td>Stored in</td>
<td>Undo segments</td>
<td>Redo log files</td>
</tr>
<tr>
<td>Protects against</td>
<td>Inconsistent reads in multiuser systems</td>
<td>Data loss</td>
</tr>
</tbody>
</table>