CHAPTER

6

Transaction and Concurrency

Objectives

- To learn the concept of database transactions.

- To learn how to use the <u>commit</u> command to commit a transaction and use the <u>rollback</u> command to rollback a transaction.

- To become familiar with the dirty read, fuzzy read, and phantom read concurrency anomalies.

- To learn how to set transaction isolation levels to read uncommitted, read committed, repeatable read, and serializable to prevent concurrency anomalies.

- To understand what is Oracle default transaction isolation level.

- To learn how to acquire locks explicitly.

- To learn how to use the <u>for update</u> clause in the select statement.

6.1 Introduction

In the preceding two chapters, you learned how to use SQL to create database objects such as tables, views, indexes, clusters, database links, and synonyms, and how to use SQL to retrieve and modify database contents. This chapter introduces how to use SQL to manage transactions and control concurrency.

6.2 Database Transactions

A *transaction* is a sequence of statements that forms a logic unit of work. All the statements in a transaction must either all succeed or all fail. Consider transferring funds between two accounts, for example, the debit from one account and credit to the other account must be in one transaction to ensure data consistency.

A transaction begins with the user’s first SQL statement and ends when it is committed or rolled back. Once a transaction ends, the new transaction starts with the user’s next SQL statement.

A transaction is committed when one of the following occurs:

- A SQL <u>commit</u> statement is executed.
- A DDL statement is executed.
- A DCL statement is executed.
- A user exits normally from SQL*Plus using the exit command.
- A user disconnects from Oracle.

A transaction is rolled back when one of the following occurs:

- A SQL <u>rollback</u> statement is executed.
- A user quits from SQL*Plus using the quit command.
- The system crashes.

When a transaction is committed, the results from the DML statements are permanently written to the database. When a transaction is rolled back, the database is restored to before the transaction started.

> NOTE: In the discussion throughout this chapter, we assume the statement has been parsed correctly. If a statement has syntax errors, the statement cannot be executed.
>
> NOTE: A transaction is committed when a DDL or DCL statement is executed regardless whether the DDL or DCL statement is successful. In Oracle, all the statements prior to the DDL or DCL statement committed first, and the DDL or DCL statement is then executed and committed in a new single statement transaction.
>
> NOTE: The successful execution of a statement is different from the successful execution of a transaction. A statement is successful if it is executed without error. A transaction is successful if all its statements are executed and committed successfully.
>
> NOTE: The failure of a statement in a transaction does not cause the transaction to be rolled back. For example, suppose an insertion

statement failed in a transaction because of the violation of the primary key constraint, the insertion statement is rolled back and the transaction continues. This is also known as the *statement-level rollback*.

You can use the savepoint statement to identify a point in a transaction to which you can later roll back instead of rolling back the entire transaction. You can set any number of save points in a transaction using the following syntax:

```
savepoint savePointId;
```

To roll back to a save point, use the following syntax:

```
rollback to savapoint savePointId;
```

For example, the following transaction has two save points:

```
savepoint before_smith;

update Faculty
set salary = 70000
where lastName = 'Smith';

savepoint before_rove;

update Faculty
set salary = 75000
where lastName = 'Rove';

select sum(salary) from Faculty;

rollback to savepoint before_rove;

update Faculty
set salary = 81000
where lastName = 'Greene';

commit;
```

NOTE: The changes made by the DML statements in a transaction are not available to other transactions until the transaction is committed.

NOTE: Oracle has an environment variable autocommit. By default, it is set to off. You can set it to on using the following command:

```
set autocommit on;
```

When autocommit is on, every SQL DML statement is automatically committed after it is executed.

******End of NOTE***

NOTE: There are several techniques used by DBMS to facilitate committing and rolling back transactions. A common approach is to use the rollback segment to store all the updates from the transaction. If committed, all the updates are written permanently to the database. If rolled back, the data in the rollback segment is discarded.

6.3 Concurrency Anomalies

A database is shared by many users. Users can request transactions on the database simultaneously. When multiple transactions execute simultaneously in the database in an uncontrolled manner, conflict may arise. Consider four scenarios with two transactions executing concurrently:

Scenario 1: Transaction T2 updates the salary of the faculty with the last name Lee and then Transaction T1 reads the salary of the same faculty before T2 is committed, as shown in Figure 6.1. It is possible that T2 may be rolled back. So the data read in T1 is not reliable. This scenario is known as the *dirty read* anomaly. i.e. a transaction reads data that has been written by another transaction that has not been committed yet.

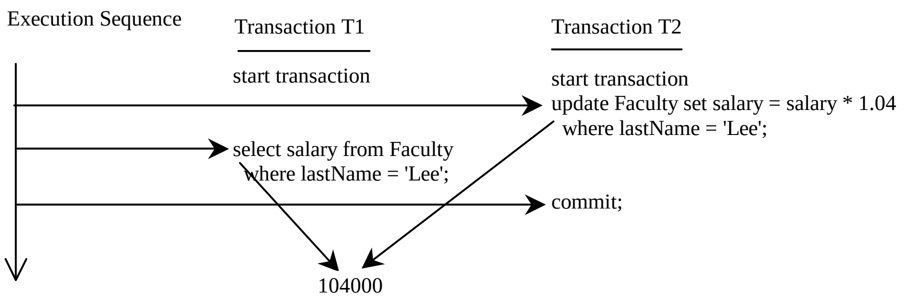

Figure 6.1

When transactions are executed concurrently, the dirty read anomaly may occur.

Scenario 2: Transaction T1 reads the salary of the faculty with the last name Lee, and Transaction T2 updates the salary of the same faculty and commits it. Transaction T1 reads the salary of the same faculty again, as shown in Figure 6.2. The two reads don’t return in the same value in Transaction T1. The data T1 is viewing has been changed by T2. This scenario is known as the *repeatable read* or *fuzzy read anomaly*. i.e. a transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data.

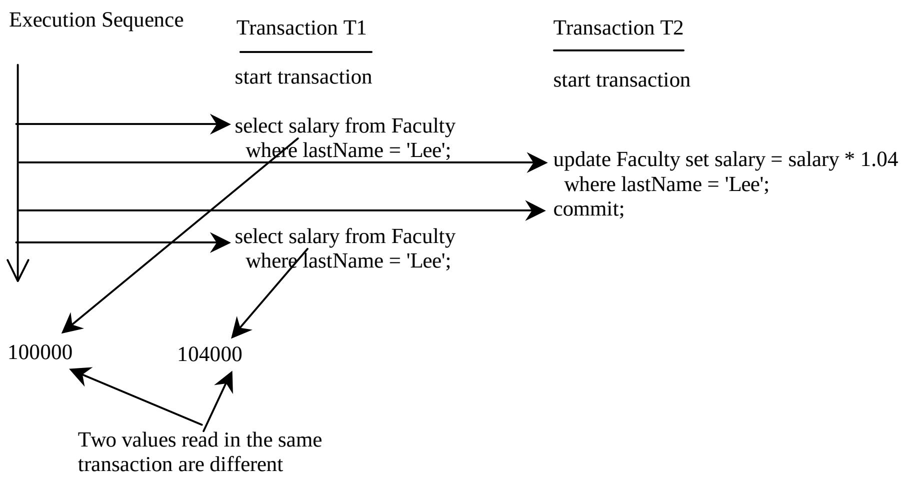
Execution Sequence
Transaction T1
Transaction T2
start transaction
start transaction
select salary from Faculty where lastName = 'Lee';
update Faculty set salary = salary * 1.04 where lastName = 'Lee';
commit;
select salary from Faculty where lastName = 'Lee';
100000
104000
Two values read in the same transaction are different

Figure 6.2

When transactions are executed concurrently, the repeatable read anomaly may occur.

Scenario 3: Transaction T1 lists the CS faculty, transaction T2 inserts a new CS faculty, and transaction T1 lists the CS faculty again using the same query, as shown in Figure 6.3. The same query in T1 returns different result. This scenario is known as the *phantom read anomaly*. i.e. a transaction re-executes a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

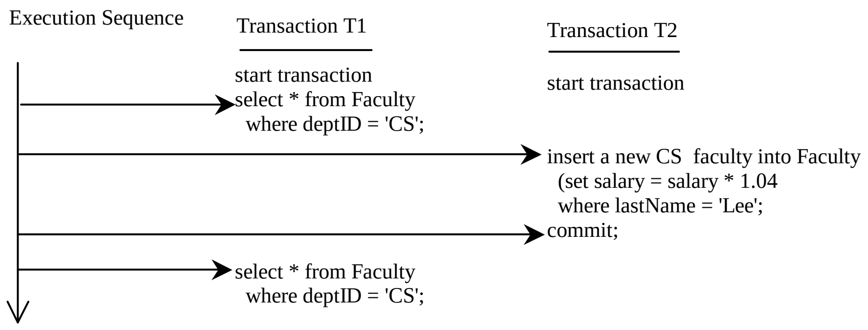
Execution Sequence
Transaction T1
Transaction T2
start transaction
select * from Faculty where deptID = 'CS';
start transaction
insert a new CS faculty into Faculty (set salary = salary * 1.04 where lastName = 'Lee';
commit;
select * from Faculty where deptID = 'CS';

Figure 6.3

When transactions are executed concurrently, the phantom read anomaly may occur.

Scenario 4: Transaction T1 increases the salary of a faculty with last name Lee by 2% and transaction T2 increases the salary of the same faculty by 4%. The executions of the transactions are interleaved, as shown in Figure 6.4.

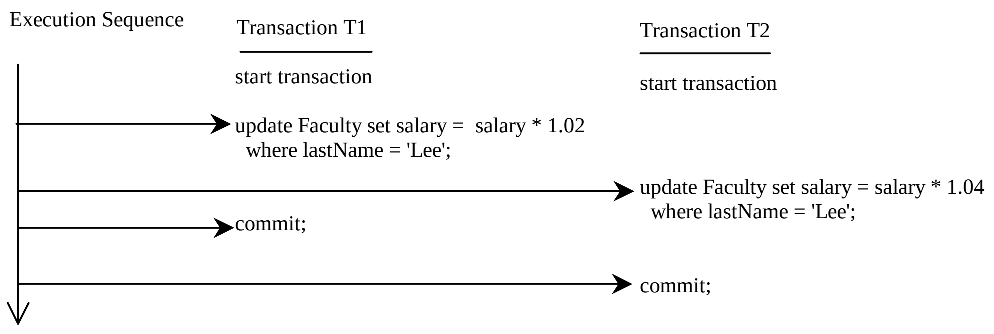
Execution Sequence
Transaction T1
start transaction
update Faculty set salary = salary * 1.02 where lastName = 'Lee';
commit;
Transaction T2
start transaction
update Faculty set salary = salary * 1.04 where lastName = 'Lee';
commit;

Figure 6.4

When transactions are executed concurrently, the override update anomaly may occur.

When transaction T1 is committed, DBMS writes a new value into the salary of Faculty 1. When transaction T2 is committed, DBMS also writes a new value into the salary of Faculty 1, which overrides the update from transaction T1. So, T1's update of the salary of Faculty 1 is in fact lost. This scenario is known as the *override update anomaly*. i.e. a transaction overrides the update of another transaction.

6.4 Preventing Concurrency Anomalies by Setting Transaction Isolation Levels

SQL defines the notion of serializability. A schedule of transactions is serializable if it is equivalent to executing the transaction in some serial order. A serializable schedule ensures database consistency. Obviously, the result of the schedules in both scenarios is different from executing the two transactions sequentially, in the order of either T1 then T2 or T2 then T1.

While serializablity is generally desirable, running many applications in this mode can seriously reduce concurrency and slow down the performance. Complete serializability could mean that a transaction cannot update the table that is being queried by another transaction. To ease the restriction, SQL92 defines four levels of transaction isolation with differing degrees of concurrency. You can set the transaction to an appropriate isolation level to prevent the abnormal scenarios while retaining some degree of concurrency.

The four transaction isolation levels defined in SQL are *read uncommitted*, *read committed*, *repeatable read*, and *serializable*.

6.4.1 Read Uncommitted Isolation Level

The read uncommitted isolation level allows a transaction to read the data that has been updated by an uncommitted transaction. Thus, all the four concurrency anomalies may occur under this level. Read uncommitted is not supported in Oracle.

6.4.2 Read Committed Isolation Level

The read committed isolation level allows a transaction to read the data that has been updated by a committed transaction. Changes made by one transaction are not seen by another concurrent transaction until the transaction that made the changes commits. Read committed isolation prevents the dirty read anomaly.

Read committed is the default Oracle transaction isolation level.

6.4.3 Repeatable Read Isolation Level

The repeatable read isolation level prevents the repeatable read anomaly by guaranteeing that a transaction reads the same data repeatedly. Oracle does not support this level, but it provides the read-only isolation level, which is more restrictive than the repeatable read isolation level. A read-only isolation level transaction permits only SQL select statements, and does not allow SQL insert/update/delete statements, but a SQL. A repeatable read transaction would also permit insert/update/delete statements. To specify a transaction to be read-only, use the following SQL statement in Oracle:

```
set transaction read only;
```

This statement establishes the current transaction as a *read-only transaction*. All subsequent queries in that transaction only see changes committed before the transaction began, except that queries do see changes made by the transaction itself.

As shown in Figure 6.5, the two salary values read from transaction T1 are the same because the transaction is set at read only, even though transaction T2 changed the same data value and committed the change.

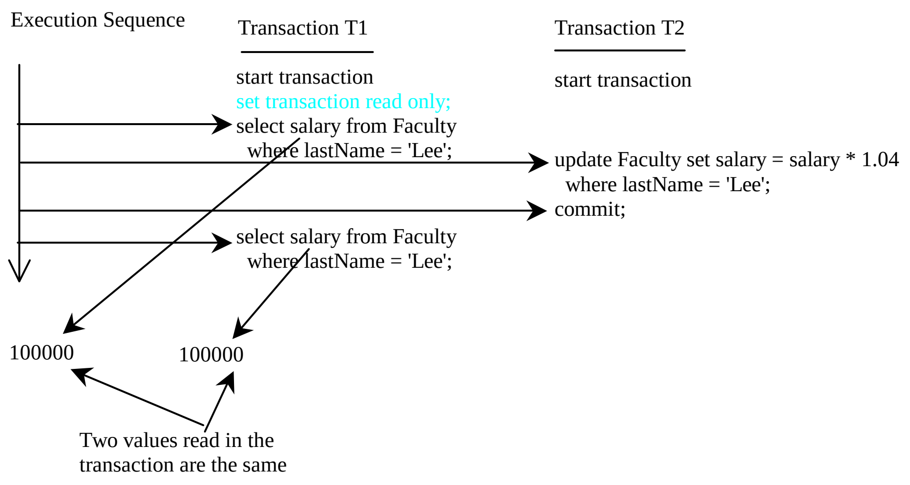

Figure 6.5

The read-only transaction prevents the repeatable read anomaly.

6.4.4 Serializable Isolation Level

The serializable isolation level can prevent all the concurrency anomalies for the transaction. You can use the following command to set a transaction isolation level serializable:

```
set transaction isolation level serializable;
```

This statement establishes the current transaction as serializable. If a serializable transaction contains a DML statement that attempts to update any resource that may have been updated in a transaction uncommitted at the start of the serializable transaction, then the DML statement fails. If a DML statement *from another transaction* that attempts to update any resource that may have been updated in a serializable transaction, the DML statement in that transaction must wait until the serializable transaction is committed.

As shown in Figure 6.6, the update statement in the serializable transaction T1 updates the salary, which causes the update salary statement in T2 to wait until T1 is committed. As shown in Figure 6.7, the update statement in the serializable transaction T1 updates the salary that has been updated in T2. This would cause T1 to be non-serializable and thus the update statement would fail.

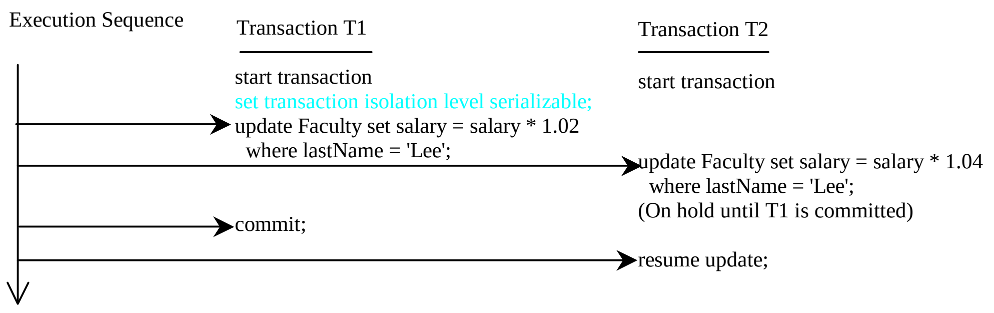
Execution Sequence
Transaction T1
Transaction T2
start transaction
set transaction isolation level serializable;
update Faculty set salary = salary * 1.02
where lastName = 'Lee';
start transaction
update Faculty set salary = salary * 1.04
where lastName = 'Lee';
(On hold until T1 is committed)
commit;
resume update;

Figure 6.6

A DML statement that attempts to update the data that has been updated in a serializable transaction must wait until the serializable transaction is committed.

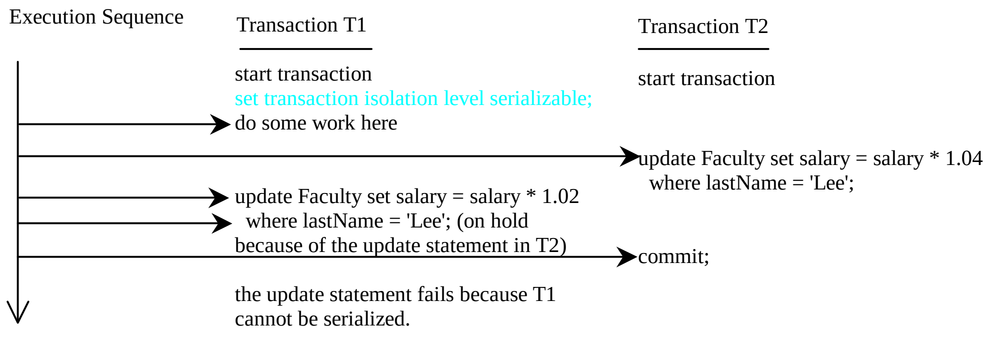
Execution Sequence
Transaction T1
Transaction T2
start transaction
set transaction isolation level serializable;
do some work here
start transaction
update Faculty set salary = salary * 1.04
where lastName = 'Lee';
update Faculty set salary = salary * 1.02
where lastName = 'Lee'; (on hold
because of the update statement in T2)
commit;
the update statement fails because T1
cannot be serialized.

Figure 6.7

A DML statement in a serializable transaction that attempts to update the data that has been updated in another transaction would fail.

NOTE: You can also set transaction isolation level serializable to avoid repeatable read anomaly, but setting read only permits more concurrency.

NOTE: You can also set transaction isolation level serializable at the session level using the following command:

```
alter session set isolation_level = serializable;
```

This command must be the first statement in a transaction, and it remains effective for all the transactions in the rest of the session.

******End of NOTE***

NOTE: If you use a set transaction statement, it must be the first statement in your transaction.

The operations performed by a set transaction statement affect only your current transaction, not other users or other transactions.

TIP: You can also use the alter session statement to set the transaction isolation level for all subsequent transactions as follows:

```
alter session set isolation level serializable;
alter session set isolation level read_committed;
```

******End of TIP***

NOTE: Oracle always guarantees that the set of data seen by a statement is consistent with respect to a single point in time and does not change during statement execution. This is known as *statement-level read consistency*.

6.4.5 Choice of Isolation Level

Now you know what read committed, read-only, and seriablizable transaction isolation levels can do. Which isolation level to set is based on your transaction requirements. Table 6.1 summarizes how these levels can be used to prevent possible anomalous scenarios.

Table 6.1

Preventing Transaction Isolation Levels

Isolation Level	Dirty Read	Fuzzy Read	Phantom Read	Override Update
Read uncommitted	Possible	Possible	Possible	Possible
Read committed	Not Possible	Possible	Possible	Possible
Repeatable read	Not Possible	Not Possible	Possible	Possible
Read-only	Not Possible	Not Possible	Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible	Not Possible

For many applications, read committed is the most appropriate isolation level. Read committed isolation can provide considerably more concurrency with a somewhat increased risk of inconsistent results due to phantoms and non-repeatable reads for some transactions.

Read-only transactions are useful for reports that run multiple queries against one or more tables while other users update these same tables.

Serializable is most restrictive. It is suitable for environments where there is a relatively low chance that two concurrent transactions will modify the same rows and the long-running transactions are primarily read-only. It is most suitable for environments with large databases and short transactions that update only a few rows. Serializable isolation mode provides somewhat more consistency by protecting against phantoms and nonrepeatable reads and can be important where a read/write transaction executes a query more than once.

6.5 Locks

Databases use locks to prevent the problems arising from concurrency. Locks are mechanisms that prevent destructive interaction between transactions accessing the same data. Setting transaction isolation levels lets Oracle automatically control concurrency. Oracle acquires locks implicitly to achieve the specified isolation level. The locks are placed at the row-level. By locking table data at the row-level, contention for the same data is minimized to increase concurrency.

You can also explicitly acquire locks on the rows and tables to have more control on the transactions. In general, there are two types of locks: *exclusive locks* and *share locks*. Only one exclusive lock can be placed on a resource (such as a row or a table); however, many share locks can be placed on a single resource. Both exclusive and share locks always allow queries on the locked resource but prohibit other activity on the resource (such as updates and deletes). Specifically, the following lock modes are supported:

- **row share (RS)**: This mode permits concurrent access to the locked table, but prohibits users from locking the entire table for exclusive access.

- **share (S)**: This mode permits concurrent queries but prohibits updates to the locked table.

- **row exclusive (RX)**: This mode is the same as row share, but also prohibits locking in the share mode. A row exclusive table lock is acquired automatically for a table modified by the insert/update/delete statement.

- **share row exclusive (SRX)**: This mode permits others to look at the rows in the table, but prohibit others from locking the table in the share mode or updating rows.

- **exclusive (X)**: This mode permits queries on the locked table, but prohibits any other activity on it.

Table 6.2 shows the compatibility of these locks.

Table 6.2

Compatibility of the locks

	RS	S	RX	SRX	X
RS	yes	yes	yes	yes	no
S	yes	yes	no	yes	no
RX	yes	no	yes	no	no
SRX	yes	yes	no	no	no
X	no	no	no	no	no

You can acquire the locks explicitly on tables and views using the following syntax:

```
lock table TableName in lockmode mode [nowait];
```

TableName may be a table name or a view name. If a view is locked, the base tables for the view are also locked. Specify nowait if you want Oracle to return control immediately on the lock command. If the specified table is already locked by another user, a message is returned to indicate the table is already locked.

For example, you can lock the Faculty table in exclusive mode using the following command:

```
lock table Faculty in exclusive mode nowait;
```

In this case, the table is locked exclusively. Other users can query the table, but cannot obtain any locks or perform other activities on the table.

> NOTE: A locked table or view remains locked until you either commit or roll it back, either entirely or a save point before you locked it.

6.6 Locking Rows in the Select Statement

Often you first view the records and then update them. You would like the selected rows to be locked so that other users cannot lock or update the selected rows before you complete your entire transaction. To do so, you can specify a for update clause in the select statement using the following syntax:

```
select column-list
from table-list
[where condition]
for update of column-list
[nowait]
```

This statement automatically acquires a share lock on the table and shared row exclusive locks for selected rows. Other user can still view the selected rows, but cannot update the rows. You can now update the rows. The locks are released when your transaction is committed.

For example, to update the salary for faculty Lee, you may first select it using

```
select salary
from Faculty
where lastName = 'Lee'
for update of salary;
```

This statement automatically acquires a share lock on the Faculty table and shared row exclusive locks for the row whose lastName is Lee in the Faculty table. The other users cannot view the Faculty table, but cannot modify the rows for faculty Lee.

> NOTE: The column-list in the for_update_clause does not mean that the locking is at the column level. The column names are just used for reference information. The locks are on the rows, not columns.
>
> NOTE: The for_update_clause must appear in the top-level select statement. It cannot be used in a subquery.

6.7 Deadlocks

Deadlocks can occur when two or more transactions are waiting for data locked by each other and none of the transactions can continue. Figure 6.8 illustrates a deadlock situation where transaction T1 is waiting for the data (Student) locked by T2 and T2 is waiting for the data (Faculty) locked by T1. Neither can continue.

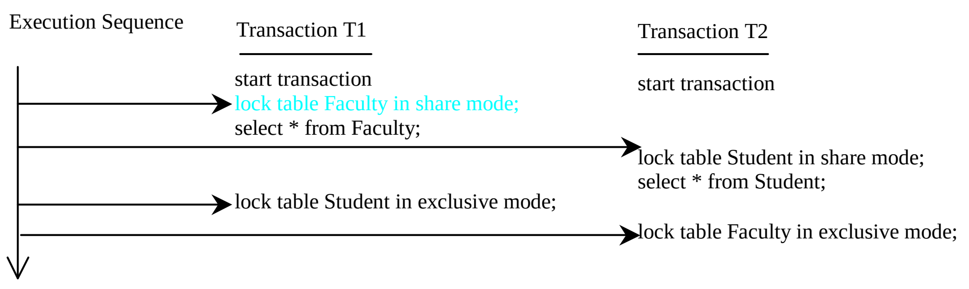
Execution Sequence
Transaction T1
Transaction T2
start transaction
lock table Faculty in share mode;
select * from Faculty;
start transaction
lock table Student in share mode;
select * from Student;
lock table Student in exclusive mode;
lock table Faculty in exclusive mode;

Figure 6.8

Transactions T1 and T2 are deadlocked.

Oracle automatically detects deadlock situations and resolves them by rolling back one of the statements involved in the deadlock, thereby releasing one set of conflicting row locks. A corresponding message also is returned to the transaction that undergoes statement-level rollback. The statement rolled back is the one belonging to the transaction that detects the deadlock.

Deadlocks can be avoided if the tables are locked in certain order. For example, in the case of Figure 6.5, if both transactions T1 and T2 lock the tables in the order of Faculty and Student, no deadlock would occur.

NOTE: Deadlocks most often occur when transactions explicitly override the default locking of Oracle. Deadlocks occur infrequently because Oracle itself does not use locks for queries.

Chapter Summary

This chapter introduced to use SQL for transaction management and concurrency control. You learned how to set the transaction level to let Oracle automatically manage concurrency, how to use the <u>lock table</u> statement to acquire <u>row share</u>, <u>share</u>, <u>row exclusive</u>, <u>share row</u> <u>exclusive</u>, and <u>exclusive locks</u> explicitly, and how to use the <u>for update</u> clause to lock selected rows for update.

Review Questions

6.1 What is a transaction? When does a transaction start and when does it end?

6.2 What happens when a transaction is committed? What happens when a transaction is rollbacked?

6.3 Does one of the following operations end a transaction?

A SQL <u>commit</u> statement is executed.
A DDL statement is executed.
A DCL statement is executed.
A user exits normally.

6.4 Does one of the following operations cause a transaction to rollback?

A SQL <u>rollback</u> statement is executed.
A user quits from SQL*Plus using the quit command.
The system crashes.

6.5 What is the purpose of using the save points?

6.6 How do you set the Oracle environment variable <u>autocommit</u> to cause every SQL statement to be committed?

6.7 What is repeatable read anomaly and override update anomaly?

6.8 How does Oracle manage concurrency by default?

6.9 How do you set transaction levels to avoid repeatable read anomaly and override update anomaly?

6.10 How do you acquire locks explicitly?

6.11 What type of locks does the <u>for update</u> clause in a <u>select</u> statement acquire?