2020

Chapter 6: Essential Aspects of Physical Design and Implementation of Relational Databases

Tatiana Malyuta
CUNY New York City College of Technology

Ashwin Satyanarayana
CUNY New York City College of Technology

Chapter 6. Transaction Management

This chapter continues the discussion of the relationship between databases and database applications started in Chapter 5. Concurrent execution of database requests can cause problems with performance and data consistency. The application has to be implemented with an understanding of these problems and the ways they are resolved by the database.

Problems of Concurrent Access and Failures

The discussion of database performance in the previous chapter concentrated on the performance of separate queries and did not consider the performance issues that could occur if queries are processed concurrently. Imagine the situation where your query is updating a row of a table, and at the same time, another user wants to make updates to the same row. If another query has to wait for your query to finish the update (and as will be shown in this chapter is a common scenario), then the performance of the second query will be definitely degraded. The waiting time component of the *response time* of the query, which was mentioned in Chapter 5, depends on concurrent queries.

Degrading performance of multiple applications working simultaneously on the database is not the only problem of concurrent query processing. Consider a simple case of concurrent actions.

Case 1.

> *A bank database keeps data about balances on checking and savings accounts of customers. John and Mary have a joint checking account, and the current balance is $100. At some moment, both of them need to withdraw money: John – $50, and Mary – $75.*

Here are two possible scenarios for the case: in the first, John and Mary perform their bank operations one after another, and in the second, they happen to do it simultaneously from different ATM machines.

Time	**Scenario 1**
1	John tries to withdraw $50. The application reads the balance on the account; the balance is $100.
2	Money is withdrawn.
3	The balance is updated to $50.
4	Mary tries to withdraw $75. The application reads the balance on the account; the balance is $50.
5	Withdrawal is rejected, because the balance cannot be negative. The balance remains $50.

Time	**Scenario 2**	
	John	**Mary**
1	John tries to withdraw $50. The application reads the balance on the account; the balance is $100.	Mary tries to withdraw $75. The application reads the balance on the account; the balance is $100.

2	Money is withdrawn.	
3		Money is withdrawn.
4	The balance is updated to $50.	
5		The balance is updated to $25.

In the second scenario, John and Mary access the same data concurrently, and, as we can see, concurrency causes problems – the correctness of the data is compromised. The resulting state of the data is inconsistent with bank business requirements.

We may imagine another scenario of concurrent access to the account, when the application performs additional checking of the account balance before updating it:

Time	**Scenario 3**	
	John	**Mary**
1	John tries to withdraw $50. The application reads the balance on the account; the balance is $100.	
2		Mary tries to withdraw $75. The application reads the balance on the account; the balance is $100.
3	Money is withdrawn.	
4		Money is withdrawn.
5	The application re-reads the balance on the account; the balance is $100.The balance is updated to $50.	
6		The application re-reads the balance on the account; the balance is $50.The balance is updated to -$25.

The concurrent scenarios show that we cannot guarantee the correctness of the results of our actions and, moreover, cannot foresee what the results would be. This definitely would be unacceptable for the businesses relying on the databases and database applications (please note that we discuss hypothetical scenarios to introduce the problems of concurrent access and then illustrate how these problems are resolved by DBMSs).

Another example shows how concurrent operations can compromise data integrity.

Case 2.

The database contains information about departments and employees:

```
Department (deptCode, deptName)

Employee (ID, emplName, deptCode)
```

Referential integrity of deptCode of the table Employee is not supported through the foreign key constraint, but in the application. Let us consider a situation when one user – John – tries to assign a new employee to the existing department '555' that does not have any employees at the time, while another user – Scott – is deleting *this department. Both John and Scott use an application that performs all the corresponding checks of data integrity.*

Time	John	Scott
1	Tries to assign an employee to the department '555' (updates the deptCode attribute row for the employee). Application ensures integrity – checks the table Department and finds the department '555'.	
2		Tries to delete the department '555' from the Department table. Application ensures integrity – checks the table Employee and does not find any employees assigned to the department '555'.
3	The deptCode attribute of the employee is updated to '555'.	The row of the department '555' is deleted from the table Department.

The resulting state of the database violates data integrity – an employee is assigned to a non-existing department '555'. Note that this would not happen if the integrity rules were specified in the database (the attribute deptCode of the table Employee defined as the foreign key to the relation Department).

The next two cases demonstrate concurrent actions on the database that do not cause any problems with the correctness of data.

Case 3.

John and Mary simultaneously check the balance of their account.

Time	John	Mary
1	John checks the account: The balance is $100.	Mary checks the account: The balance is $100.
2	The balance remains $100.	The balance remains $100.

Case 4.

Scott has a checking account with the balance of $200 in the same bank. He needs to withdraw money simultaneously with John.

Time	John	Scott
1	John tries to withdraw $50. The application reads the balance on the account; the balance is $100.	Scott tries to withdraw $75. The application reads the balance on the account; the balance is $200.
2	Money is withdrawn.	Money is withdrawn.
3	The balance is updated to $50.	The balance is updated to $125.

Whichever scenario we choose for execution of actions of these two situations, the resulting state of data always will be correct. Concurrency does not create problems when: (a) Concurrent operations read the same data and (b) Concurrent operations access different data.

The above examples show that in some cases simultaneous actions of several users can lead to incorrect or inconsistent database states (we understand the correctness of a database state as its correspondence to business rules). Does this mean that concurrent execution of transactions should not be allowed? On the other hand, we understand that concurrency significantly increases the performance of database applications, e.g. if in the above examples, operations were executed serially (one after another), the throughput of the database would be lower or, in other words, the execution of all operations would require more time. Besides, as we see from Cases 3 and 4, for some operations concurrent execution does not cause database inconsistencies. Operations involving concurrent executions that might result in incorrect states of the database are called *conflicting*.

Concurrent execution of multiple operations often is the only possibility for the database to satisfy performance requirements. For example, even with the best possible performance of the bank application, if customers' requests were processed serially, this would cause many users to wait for their turn to perform a bank operation, which, of course, is unacceptable for this business. Hence databases are required to have mechanisms for executing operations concurrently. Concurrent mechanisms have to recognize conflicting operations and regulate their simultaneous execution, so that the resulting state of the database is consistent with business rules.

The consistency of a database can be also endangered by database failures.

Case 5.

John also has a savings account, and he wants to transfer $50 from his savings account to his checking account. Before the transfer, his savings and checking balances are $100.

Here is one of the possible scenarios:

Time	Scenario
1	John tries to transfer $50. The application reads the balance on the savings account; the balance is $100.
2	$50 are withdrawn from the savings account. The savings balance is updated to $50.
3	The database (or the application) fails.
4	…
5	After the database (or the application) is recovered, the savings balance is $50, and the checking balance is $100.

In this scenario, the incorrect state of the database happened because of the failure that interrupted the sequence of operations. To preserve data consistency, the database must be able to recover from failure either by finishing the sequence of operations, or by undoing the withdrawal from the savings account and returning to the initial state.

Let us summarize the discussion of the cases. A database state may become inconsistent with business rules if:

1. *Conflicting* operations are executed *concurrently*. Therefore, to enable concurrent execution, the database has to recognize conflicting concurrent operations and organize their execution to prevent inconsistency. The

organization of concurrent processing has to be aimed at preserving data consistency and achieving appropriate performance.

2. The *sequence* of operations is interrupted by *failure*. To prevent inconsistency, the database must be able to recover from failure to a consistent state either by undoing the performed operations and returning to the initial state before the sequence started, or by finishing the sequence.

For each sequence of operations on the database, business rules define the correct final state of the database. For example, in *Case 5* we see that money transfer operations have to result in the balance of one account being decreased by $50 and the balance of another increased by $50. The resulting state of the database has to be consistent with the logic of operations and business rules. Figure 1 shows two sequences of operations T1 and T2, which start from states $State_1$ and $State_2$ and must take the database into states $State_3$ and $State_5$, respectively. Any other final state of the database is incorrect ($State_4$) and has to be prevented by the database. In case the correct final state cannot be achieved, the database must be returned to the previous correct state, from which the sequence of operations started.

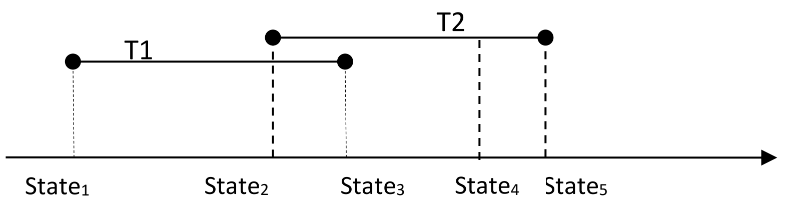


Figure 1. States of the database.

The Concept of a Transaction

In the cases of the previous section, sequences of database actions that implemented the logic of business processing were executed incorrectly either because of the harmful impact of concurrent operations or because some operations of the sequence were not executed. Hence, the concept of a transaction, as a *logical unit* of work in the database, was introduced.

Transactions support consistent database computing.

A transaction is a sequence of operations on the database (reads, writes, and calculations), which takes the database from one consistent state to another, and if this is not possible, returns the database into the initial consistent state.

A transaction is different from a separate operation (query). The operations of a transaction do not act independently – each operation belongs to the transaction. The successful execution of a particular operation does not guarantee that the result of this operation will be applied to the database – this happens only if all other operations of the transaction are executed successfully. If a transaction includes only one operation, then the success of the transaction depends on the success of this single operation.

The transaction is a tool of *consistent and reliable* database processing. By reliability we mean that a system is resilient to various types of failures and is capable of *recovering* from them to a consistent database state without data loss.

This chapter discusses transactions and database consistency under the conditions of concurrent access to data. The next chapter is dedicated to the role of transactions in the recovery of databases.

The consistency of the database depends not only on how the system processes transactions, but also on the correctness of the transactions. If the database programmer incorrectly implements the business logic in a transaction, then the database will always become inconsistent with the business rules. For example, if a money transfer transaction is implemented in such a way that a particular amount of money is deposited to one account and a different amount is withdrawn from another account, then the database will become inconsistent even if the transactions are executed serially and there are no failures. In this discussion, we assume that the transactions correctly implement the business logic.

The support of data consistency can affect the performance of some transactions. When designing transactions, database programmers have to utilize transaction management features of the DBMS in such a way that the consistency of data is preserved, and the performance of concurrently executed transactions is not compromised.

The sequence of operations in a transaction is finished by the commands COMMIT or ROLLBACK. Committing the transaction means that all operations were executed successfully and there was no threat of inconsistency caused by other transactions. Rolling back means that the transaction is aborted and the results of all performed operations are disregarded. The transaction explicitly defines consistent states of the database – the next consistent state is achieved after a COMMIT, and the database is returned to the previous consistent state in the case of a ROLLBACK. In some DBMSs the beginning of a transaction is defined by a special statement, e.g. BEGIN TRANSACTION in MS SQL Server. In other DBMSs, including Oracle, the new transaction is immediately started after COMMIT or ROLLBACK.

The following is an example of using COMMIT and ROLLBACK in Oracle on a newly created table:

```
     CREATE TABLE test (f1 NUMBER);
```

`SELECT * FROM test;`
0 rows returned
`INSERT INTO test VALUES (1);`
 	
`COMMIT;`
	The first transaction inserts a record into the table and commits.

`SELECT * FROM test;`
1 row returned (with f1 equal 1)
 	
`INSERT INTO test VALUES (2);`
 	
`SELECT * FROM test;`
2 rows returned (with f1 equal 1 and 2)
`ROLLBACK;`
	The SELECT statement starts the second transaction and shows that the INSERT of the first transaction is permanent.
	
	The transaction inserts the second record, and the following SELECT shows it.
	
	The transaction performs ROLLBACK.

`SELECT * FROM test;` *1 row returned (with f1 equal 1)*	The third transaction starts after the ROLLBACK. The result of the action of the second transaction has not been recorded in the database – the database returned into the initial state of the second transaction.

Let us consider organizing the actions of *Case 5* in a transaction, but in a different order than before: first, we deposit $50 into the checking account, and then withdraw $50 from the savings account. Also, assume that the balance of the savings account has been changed to $30.

Time	Scenario
1	John tries to transfer $50.
2	$50 is deposited on the checking account; the balance of the checking account is updated to $150.
3	The application reads the balance on the savings account; the balance is $30.
4	Money cannot be withdrawn from the savings account.
5	The depositing of $50 on the checking account has to be undone.
6	The transaction is finished with ROLLBACK.

The above scenario shows that the design of a transaction plays an important role in the database performance. Because depositing is executed first, without checking that the money transfer is possible, the system wastes resources on executing this operation and then rolling it back.

The following scenario shows a more efficient transaction design; it is executed successfully because the initial savings balance is changed to $100:

Time	Scenario
1	John tries to transfer $50. The application reads the balance on the savings account; the balance is $100.
2	$50 is withdrawn from the savings account. The savings balance is updated to $50.
3	The balance of the checking account is updated to $150.
4	The transaction is finished with COMMIT, and the new balances on the accounts become permanent.

For system failures during the execution of concurrent transactions, most DBMSs roll back all uncommitted transactions if the database processing was interrupted abnormally. It is important to understand that if operations were not organized in transactions, the system would not know which operations to roll back. Transaction processing, failures and database recovery are also discussed in the next chapter.

Let us take a closer look at the problems of concurrent execution and how these problems can be prevented or resolved with the help of transactions.

Assuming that database processing is reliable, serial execution of transactions does not cause any inconsistencies. We will check the correctness of the concurrent execution of transactions by comparing the results with the results of the serial execution of the same transactions – the results that we know are correct and the ones we expect. The following problems – they are called phenomena – are recognized in transaction processing. Please note that the following scenarios are not the only possible scenarios of concurrent execution of the involved transactions – these scenarios were specifically chosen to illustrate the phenomena. Even if not every data processing situation results in a phenomenon, the system needs to recognize the problematic situations and prevent the phenomena from happening.

Lost Update.

Initially x = 100. The transaction T1 adds 1 to x; the transaction T2 subtracts 1 from x. Serial execution of the transactions results in x = 100. Concurrent execution of the transactions in the following scenario results in x = 99.

Time	T1	T2
1	Read (x) *x = 100*	
2	x ← x + 1	Read (x) *x = 100*
3	Write (x) *x = 101*	x ← x - 1
4	Commit	Write (x) *x = 99*
5		Commit

The update of the transaction T1 was not seen by the transaction T2.

Dirty Read.

Initially x = 100. The transaction T1 adds 1 to x and then rolls back; the transaction T2 subtracts 1 from x. Serial execution of the transactions results in x = 99. Concurrent execution of the transactions in the following scenario results in x = 100.

Time	T1	T2
1	Read (x) *x = 100*	
2	x ←x + 1	
3	Write (x) *x = 101*	
4		Read (x) *x = 101*
5	Rollback	x ← x – 1
6		Write (x) *x = 100*
7		Commit

The dirty read happens because T2 is dependent on the changes performed by T1. If T1 is rolled back, T2 continues with dirty, "non-existing" data. To prevent the inconsistency caused by the dirty read, T2 should have been rolled back.

Fuzzy (non-repeatable) read.

Initially x = 100. The transaction T1 reads x two times; between the reads of the transaction T1, the transaction T2 subtracts 1 from x. In serial execution, T1 will always read the same value – 100 or 99 –

depending on the order of the transactions. For concurrent execution, reads of T1 are inconsistent (non-repeatable).

Time	T1	T2
1	Read (x) *x = 100*	
2		Read (x) *x = 100*
3	…	x ← x – 1
4		Write (x) *x = 99*
5	Read (x) *x = 99*	…
6	Commit	Commit

Phantom.

The transaction T1 reads 100 rows, which satisfy some condition F, and then updates these rows. However, because the transaction T2 deletes one of the rows while T1 processes the read records, T1 does not write 100 rows as it intended, but only 99 – it updated the deleted phantom row. If T2 added a new row, T1 would have a new phantom row that it did not update. Phantoms are similar to fuzzy reads.

Time	T1	T2
1	Read all x_i for which $F(x_i)$ = TRUE *100 rows read*	Read (x_j)
2		Delete (x_j)
3	Update all x_i *100 rows updated*	
4	Write all x_i for which $F(x_i)$ = TRUE *99 rows written*	
5	Commit	Commit

Another example of phantom would be if T1 in the above scenario was finding the sum of all x_i (moment 3). Then, the result of T1 will include the record that has been deleted.

Properties of Transactions

The examples of the phenomena show that just organizing operations into transactions does not protect the database from inconsistency. A transaction itself cannot foresee and prevent the above phenomena; that is why the DBMS contains a transaction manager that coordinates the concurrent execution of transactions. The consistency of the database is defined by how the transaction manager supports properties of transactions. The following are the properties of transactions defined by the ANSI/ ISO SQL standards (http://webstore.ansi.org):

- Atomicity
- Consistency
- Isolation
- Durability

Atomicity defines the transaction as a unit of processing: either all actions of the transaction are completed or none of them. If the transaction fails, then the database has to be recovered either by undoing the performed actions and returning into the starting consistent state or by completing the remaining actions and going to the next consistent state.

Consistency defines the degree of correctness of transactions or, in other words, what phenomena, if any, can occur during transaction processing. When we discuss the correctness of the transaction, we mean the correct concurrent execution of the transaction by the system, assuming that when executed separately, the transaction is correct. ANSI defines four levels of consistency of concurrent transactions:

1. Transaction T sees degree 3 consistency if:
   - T does not overwrite dirty data of other transactions.
   - T does not commit any writes until it completes all its writes.
   - T does not read dirty data of other transactions.
   - Other transactions do not dirty any data read by T before it completes.

2. Transaction T sees degree 2 consistency if:
   - T does not overwrite dirty data of other transactions.
   - T does not commit any writes until it completes all its writes.
   - T does not read dirty data of other transactions.

3. Transaction T sees degree 1 consistency if:
   - T does not overwrite dirty data of other transactions.
   - T does not commit any writes until it completes all its writes.

4. Transaction T sees degree 0 consistency if:
   - T does not overwrite dirty data of other transactions.

Transaction consistency is maintained by the transaction manager. In the above scenarios of phenomena, inconsistencies are caused by *uncontrolled* executions of transactions that are reading and writing dirty data of other transactions.

Isolation defines how the transaction is protected from the impact of operations of other transactions. Insufficient isolation does not provide for high consistency. If there is not enough isolation, transactions can be executed incorrectly, or have to be aborted and rolled back to avoid inconsistency.

ANSI defines four levels of isolation:

- *Read uncommitted.* The transaction can read uncommitted changes of other transactions. With this level of isolation, all phenomena are possible, as we could see from the previous examples.
- *Read committed.* The transaction can read only committed changes of other transactions. With this level of isolation, fuzzy reads and phantoms are possible.
- *Repeatable read.* The transaction performs repeatable reads regardless of any committed changes of other transactions. With this level of isolation, only phantoms are possible.

- *Anomaly serializable.* The transaction is executed as if it were the only one in the database. With this level no phenomena happen.

In the examples of phenomena introduced above, transactions are not isolated enough – they interfere with each other while performing operations on the same data. Therefore, these transactions are not consistent enough. For the lost update T2 is allowed to read data before T1 has committed it; in the dirty read, T2 reads data that was previously written and not committed by T1; in the fuzzy read, T1 reads uncommitted data of T2, and T2 writes data, which was previously read by T1; in the phantom T2 changes the number of data values while T1 is trying to process and update the data.

Durability means that when the transaction is completed, the changes cannot be undone.

We may say that Atomicity and Durability are trivial transaction properties as they follow from the definition of the transaction. We will focus on Consistency and Isolation (that defines Consistency) and the way they are maintained in general and in Oracle in particular.

Isolation Level of Transactions in Oracle

The following simple test allows us to determine the isolation level of transactions in the Oracle DBMS. The table test was populated with 100 rows containing sequential numbers in both fields (to generate the numbers we use and illustrate the sequence object of the Oracle database and a simple insert query that uses one of the existing database objects – the view user_objects from the data dictionary):

```
CREATE TABLE test (f1 NUMBER, f2 NUMBER);

CREATE SEQUENCE seq_test;

INSERT INTO test
      (SELECT seq_test.NEXTVAL, seq_test.NEXTVAL
      FROM user_objects
      WHERE ROWNUM < 101);
```

Time	T1	T2
1	`SELECT COUNT(*) FROM test;` *Returns COUNT(*) = 100*	
2		`DELETE FROM test WHERE f1 <= 50;` *50 rows deleted*
3	`SELECT COUNT(*) FROM test;` *Returns COUNT(*) = 100*	
4		`COMMIT;`
5	`SELECT COUNT(*) FROM test;` *Returns COUNT(*) = 50*	
6	`COMMIT;`	
7		`UPDATE test SET f2 = 1 WHERE f1 = 51;` *1 row updated*
8	`SELECT f2 FROM test WHERE f = 51;` *Returns f2 = 51*	

9		`COMMIT;`

10	`SELECT f2 FROM test`
	`WHERE f1 = 51;`
	Returns f2 = 1	

This test shows that:

- There are phantoms: T1 reads 100 rows at moment 3 (after some rows were deleted by another transaction) and 50 rows at moment 5.
- There are fuzzy reads: T1 reads the value of the field of a particular row at moment 8 and gets 51, then it reads the value of the field of the same row at moment 10 and gets 1.
- There are no dirty reads: at moment 8, T1 does not see the uncommitted change of T2, which was performed at moment 7.

We can conclude that the isolation level in Oracle is READ COMMITTED.

Serializability Theory

The safest way to execute transactions is to execute them serially; however, most database applications need better performance than could be provided by serial execution. Besides, in many cases concurrent access does not endanger the consistency.

All previous examples show that *uncontrollable* concurrent access to data can cause serious problems. Therefore, transaction management is directed at finding some *regulated* way of concurrent execution of transactions, which would be a trade-off between data consistency provided by the serial execution and better performance provided by the concurrent execution.

To regulate the execution of transactions, the database builds schedules. A schedule is an interleaved order of execution of operations of concurrent transactions. For a pair of transactions, the system can generate multiple schedules. Let us consider the following transactions:

T1: $\{O_{11}, O_{12}, \ldots, O_{1n}\}$

T2: $\{O_{21}, O_{22}, \ldots, O_{2m}\}$

and one of the possible schedules of their concurrent execution:

S1: $\{O_{11}, O_{21}, \ldots, O_{1i}, O_{2j}, \ldots, O_{1n}, O_{2j+1}, \ldots, O_{2m}\}$

The database appears in an inconsistent state only when transactions concurrently access the same data. Such transactions are called conflicting. Not every pair of concurrent accesses to a particular data is conflicting:

- The read of one transaction never conflicts with the read of another transaction (*Case 3*).
- The read of one transaction conflicts with the write of another transaction (*Case 1*, phenomena).
- The write of one transaction conflicts with the write of another transaction (lost update phenomenon).

For conflicting operations, the order of these operations (the schedule) is important and defines the result. Scenarios for the *Case 1* demonstrate the correct result for the first schedule (serial execution), and different incorrect results for the two concurrent schedules.

Case 6.

Let us combine the cases 1 and 4: John transfers $50 from the savings account to the checking account and then withdraws $50 from the checking account. Mary withdraws $75 from the checking account.

Consider several different scenarios:

Time	**Scenario 1**	
	John	**Mary**
1	John tries to transfer $50. The application reads the balance on the savings account; the balance is $100.	Mary tries to withdraw $75 from the checking account: The application reads the balance on the account; the balance is $100.
2	$50 is withdrawn from the savings account.	$75 is withdrawn from the checking account.
3	The savings balance is updated to $50.	The checking balance is updated to $25.
4	$50 is deposited on the checking account.	
5	The checking balance is updated to $75.	
6	John tries to withdraw $50. The application reads the balance on the checking account; the balance is $75.	
7	$50 is withdrawn.	
8	The checking balance is updated to $25.	

Time	**Scenario 2**	
	John	**Mary**
1	John tries to transfer $50. The application reads the balance on the savings account; the balance is $100.	
2	$50 is withdrawn from the savings account.	
3	The savings balance is updated to $50.	
4	$50 is deposited into the checking account.	
5	The checking balance is updated to $150.	
6	John tries to withdraw $50: The application reads the balance on the savings account; the balance is $150.	
7	$50 is withdrawn.	Mary tries to withdraw $75 from the checking account: The application reads the balance on the account; the balance is $150.
8	The checking balance is updated to $100.	$75 is withdrawn
9		The checking balance is updated to $75.

The first scenario of concurrent execution of John and Mary's transactions results in the same database state as the serial execution would have. Different schedules that result in the same database state are called conflict equivalent. Concurrent schedules that are conflict equivalent to serial schedules are called serializable.

The goal of concurrent transactions management is to build serializable schedules of execution of transactions, combining the benefits of performance of concurrent execution with the correctness of serial execution.

Here is the example with two transactions which perform the following actions:

T1
Read (x)
x ← x + 1
Write (x)
Commit

T2
Read (x)
x ← x + 1
Write (x)
Commit

We can build different schedules for execution of these transactions: serial schedules, in which operations of one transaction are followed by operations of another transaction, and concurrent schedules, in which operations of one transaction are interleaved with operations of another transaction. In the following schedules, the operations of the transactions are defined with their first letters: R is for Read, W – for Write, U – for Update, C – or Commit, and RB – for Rollback, and indexes define the transaction to which the operation belongs:

S1 = {R1, U1, W1, C1, R2, U2, W2, C2}

S2 = {R2, U2, W2, C2, R1, U1, W1, C1}

S3 = {R1, U1, W1, R2, U2, W2, C1, C2}

S4 = {R1, U1, R2, W1, U2, W2, C1, C2}

S5 = {R2, U2, R1, W2, U1, W1, C2, C1}

If before the execution of the transactions x = 0, then the result of the first three schedules is x = 2, and the result of the last two schedules is x = 1.

We have two sets of the conflict equivalent schedules: (S1, S2, S3) and (S4, S5). Schedules S1 and S2 are serial, the schedule S3 is conflict equivalent with serial schedules and, therefore, is serializable. Schedules S4 and S5 are conflict equivalent with each other, but are not equivalent to serial schedules, and, therefore, are non-serializable. Serializable schedules take the database into the consistent state, while non-serializable schedules result in an inconsistent database state.

Here is another example of transactions:

T1
Read (x)
x ← x + 1
Write (x)
Read (y)
y ← y + 1
Write (y)
Commit

T2
Read (x)
x ← x * 2
Write (x)
Read (y)
y ← y * 2
Write (y)
Commit

For two transactions Ti and Tj, Ti → Tj means that Ti is followed by Tj. Initially, x = 10 and y = 10. For T1 → T2, the result of transactions is x = 22 and y = 22. For T2 → T1, the result is x = 21 and y = 21.

Let us discuss several possible concurrent schedules. Schedule S1 results in x = 22 and y = 22; it is serializable for T1 → T2:

> S1 = {R1(x), U1(x), W1(x), R2(x), U2(x), W2(x), R1(y), U1(y), W1(y), R2(y), U2(y), W2(y), C1, C2}

Schedule S2 results in x = 21 and y = 21; it is serializable for T2 → T1:

> S2 = {R2(x), U2(x), W2(x), R1(x), U1(x), W1(x), R2(y), U2(y), W2(y), R1(y), U1(y), W1(y), C1, C2}

Schedule S3 results in x = 20 and y = 22; it is not serializable:

> S3 = {R1(x), U1(x), R2(x), W1(x), U2(x), W2(x), R1(y), U1(y), W1(y), R2(y), U2(y), W2(y), C1, C2}

The next example shows the schedule for the concurrent execution of three transactions:

T1
Read(x)
x ← x + 1
Write(x)
Commit

T2
Read(x)
x ← x + 1
Write(x)
Read(y)
y ← y + 1
Write(y)
Read(z)
Commit

T3
Read(x)
Read(y)
Read(z)
Commit

For x, y, z that initially are equal to 0, the serial schedule T2 → T1 → T3 results in x = 2, y = 1, z = 0.

The following schedule is serializable for T2 → T1 → T3:

> {R2(x), W2(x), R1(x), W1(x), C1, R3(x), R2(y), W2(y), R3(y), R2(z), c2, R3(z), C3}

Concurrency Control Approaches

Approaches for managing concurrent transactions i.e. building serializable schedules of their execution – can be divided into two groups:

- *Locking.* Locking is a mechanism, which controls access to data for conflicting operations in transactions. To sustain serializable execution of transactions, locking causes some operations to be postponed or be denied access to data.

- *Scheduling.* Scheduling, or timestamping is another mechanism that performs ordering of transactions' operations in a serializable manner based on rules or protocols.

Pessimistic and Optimistic Approaches to Transaction Management

Approaches to transaction management are also classified into pessimistic and optimistic.

Pessimistic approaches assume that the conflict between concurrent transactions is possible and try to prevent inconsistency of the database as early as possible.

Optimistic approaches, on the other hand, are based on the assumption that the conflict is unlikely. These approaches try to schedule operations in transactions and postpone checking of the correctness of the schedule until its execution. If executing an operation violates the rules of consistency, then the transaction, which contains the operation, is aborted.

Locking

Case 4 showed that concurrent transactions which access different data do not cause inconsistency. We say that these transactions are not conflicting for the data. Execution of non-conflicting transactions does not require special rules; and further discussion will be provided for conflicting transactions, that is, transactions concurrently accessing the same data.

Case 3 showed that if one transaction reads data, another transaction can read the same data without compromising data consistency. Two (or more) transactions can read the same data concurrently, or we say, can share reading access to data.

The read operation does not conflict with the read operation of another transaction.

The example with the fuzzy read shows that if a transaction reads data, it cannot allow another transaction to write this data because of possible inconsistency. The example with the dirty read shows that if a transaction writes data, it cannot allow another transaction to read it. The example with the lost update shows that if a transaction writes data, it cannot allow another transaction to write the same data.

The read operation conflicts with the write operation of another transaction. The write operation conflicts with the write operation of another transaction.

Two transactions cannot share read – write or write – write access to the same data. In this case, only one transaction can get exclusive access to the data.

Types of Locks

Depending on the type of access to the data, the transaction leaves a trace (or a mark), which shows what type of access to this data is allowed for concurrent transactions. This trace is called a lock. There are two types of locks: *Shared* and *Exclusive*.

If a data item receives a shared lock from one transaction, this item can be read by other transactions, but none of the other transactions can write in this data item. If a data item receives an exclusive lock from a particular transaction, this transaction can read this data item and write it, but other transactions can neither read nor write it. In other words, two reads can be performed concurrently on a data item, but a read and a write or a write and a write cannot.

If a transaction involves reading data, it tries to obtain a read lock (shared lock) on the data. This is possible if the data item does not have an exclusive lock of another transaction. However, the data item can have a shared lock of another transaction or not have any lock.

If the transaction involves writing data, it tries to obtain a write lock (exclusive lock). This is possible, if the data item does not have a lock of another transaction. If the data item has a shared or exclusive lock of another transaction, the transaction cannot get an exclusive lock on the data item.

For example:

Time	**T1**	**T2**
1	Read (x) *Applies a shared lock to x.*	
2		Read (x) *Is allowed because T1 has a shared lock of x. Applies a shared lock.*
3	Write (x) *Cannot apply an exclusive lock because of the shared lock of T2.*	

And another example:

Time	**T1**	**T2**
1	Write (x) *Applies an exclusive lock to x.*	
2		Read (x) *Is not allowed to apply the shared lock because of the exclusive lock of T1.*

For a data item, more than one transaction can hold a shared lock, but only one transaction can have an exclusive lock.

Some systems allow the transaction to upgrade its shared lock to an exclusive lock or downgrade the exclusive lock to a shared lock. For example, if the transaction first reads a data item and applies a shared lock, and then writes the data item, then the shared lock of the data item is changed to an exclusive lock. This could happen if other transactions do not have any lock of this data item.

Releasing Locks

A transaction locks a data item when it accesses it. This makes the item completely or partially (depending on the type of the lock) unavailable to other transactions. When should the lock be released to allow other transactions to proceed with access to the data item?

Let us assume that the transaction locks an item when it performs an operation and releases the lock immediately after the operation is finished to make the item available to other transactions. Then the transaction proceeds with its other operations. This approach does not postpone other transactions for long, but does it work for data consistency?

The following schedule represents our example with dirty reads:

S = {R1, U1, W1, R2, RB1, U2, W2, C2}

This schedule with locks will look like (a shared lock is shown as SL, an exclusive lock – as EL, and a lock release as LR):

S = {*SL1*, R1, *LR1*, U1, *EL1*, W1, *LR1*, *SL2*, R2, *LR2*, RB1, U2, *EL2*, W2, *LR2*, C2}

The result of this schedule is the same as it was without locking – we have a dirty read. Releasing the lock of an operation immediately after the operation is finished does not solve the inconsistency problem because it does not isolate the transactions and allows them to interfere with each other.

Consistent concurrent transaction management uses a *two-phase locking* mechanism (2PL). The rules of the 2PL mechanism are:

- No transaction can request a lock after it releases a lock.
- A transaction cannot release a lock until it is sure that it will not need another lock.

With the 2PL mechanism, every transaction is executed in two phases: the growing phase, when it accesses data items and locks them, and the shrinking phase, where the transaction releases the locks. At a certain point called the *lock point, when* the transaction is finished with locking, but has not released the locks yet, there are two possibilities: to release locks one by one, or to release all locks at one time when the transaction is terminated.

The two-phase locking mechanism with a "gradual" shrinking phase (Figure 2) reorganizes the schedule for the dirty read example in the following way:

S = {*SL1*, R1, U1, *EL1*, W1, *LR1*, *SL2*, R2, RB1, U2, *EL2*, W2, *LR2*, C2}

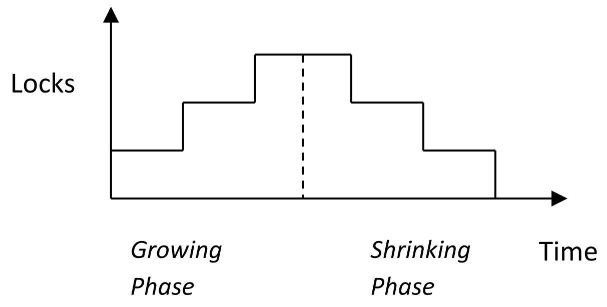


Figure 2. Gradual shrinking of a transaction.

This schedule is not serializable and it still results in a dirty read. After the first transaction has finished all its operations, it started releasing the locks. The second transaction got access to the data item before the first transaction was finished, read the data item, and because it was before the end of the first transaction, it was a dirty read. The first transaction rolled back and created the dirty read phenomenon in the database. Note that in this case the first transaction upgraded the lock on the data item from the shared to exclusive. This was possible because no other transaction was holding any lock on the data item.

Locking protects database consistency if the shrinking phase is implemented as an "immediate" lock release for all locks when the transaction is finished (Figure 3).

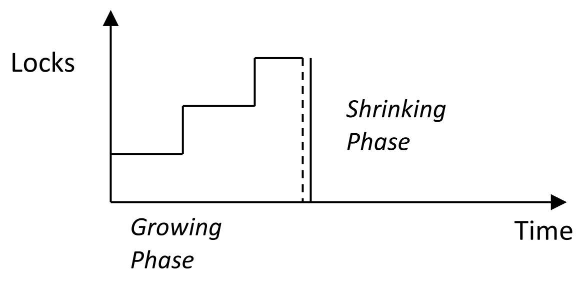


Figure 3. Immediate shrinking of a transaction.

All locks of a transaction are released at the same time when the transaction is finished.

The schedule for the dirty read example is changed. Because the lock release of the first transaction is performed when the transaction is finished, the second transaction cannot apply the shared lock where it had it before. The second transaction waits for the first transaction to release the lock. Actually, the locking forces the transactions to be executed serially:

S = {*SL1*, R1, U1, *EL1*, W1, *LR1*, RB1, *SL2*, R2, U2, *EL2*, W2, *LR2*, C2}

Here is an interesting example about serializability and locking: locking enforces serializability, however, not every serializable schedule is possible with locking:

T1
Read(x)
x ← x + 1
Write(x)
Read(x)
y ← y * 10
Write(y)
Commit

T2
Read(x)
Commit

T3
Read(y)
Commit

For x = 1 and y = 1 and T3 → T1 → T2 we will get x = 2, y =10.

The following schedule (locks are implied) is serializable with T3 → T1 → T2, but it would not be allowed by the 2PL mechanism (because T1 will need a lock on y after it is finished with processing x, it cannot release the lock on x and make x available to T2 and T3). The operation of T2, which is not allowed by locking, is shown in italic.

S= {R3(x), R1(x), U1(x), W1(x), *R2(x)*, R3(y), R1(y), U1(y), W1(y), C1, C2, C3}

In general, locking supports serializability of the concurrent execution of transactions in the following way: for two transactions

T1: {O_{11}, O_{12}, …, O_{1n}}

T2: {O_{21}, O_{22}, …, O_{2m}}

the system builds a concurrent schedule until it encounters an operation which conflicts with an already executed operation of another transaction:

S: {O_{11}, O_{21},…, O_{1i}, O_{2j},…, O_{1i+1}

The next operation O_{1i+1} of T1 cannot be executed because it conflicts with the operation O_{2j} of T2 (it means that O_{2j} holds a lock on the data item requested by O_{1i+1}). Execution of T1 cannot be continued and is postponed until the end of T2, when T2 releases all its locks:

S: {O_{11}, O_{21},…, O_{1i}, O_{2j}, O_{2j+1}, O_{2m}, O_{1i+1} …, O_{1n}}

Let us revise the examples of phenomena under the locking approach.

Lost Update:

Time	T1	T2
1	Read (x)	
2	x ← x + 1	Read (x)
3	Write (x) *Waits because T2 holds a shared lock (read at moment 2).*	x ← x – 1
4		Write (x) *Waits because T1 holds a shared lock (read at moment 1).*
5	*Deadlock: one of transactions has to be canceled.*	
	Transaction is interrupted, the write operation is cancelled.	*Waits*
6	Rollback	
7		*Write is performed*
8		Commit

This example demonstrates the disadvantage of the locking approach: it can cause situations where one transaction is waiting for another transaction to release a lock, while the other transaction is waiting for the first transaction to release its lock. Such situations are called *Deadlocks.* A deadlock cannot be resolved by the transactions themselves. The database transaction manager detects a deadlock situation and aborts one of the operations that caused the deadlock: in the above example it was the Write of the transaction T1. As one of the operations of the transaction T1 is cancelled, the transaction has to be rolled back (remember the atomicity property!).

***Dirty Read*:**

Time	T1	T2
1	Read (x)	
2	x ←x + 1	
3	Write (x)	

4	…	Read (x) *Waits because T1 holds an exclusive lock (write at moment 3)*
5	Rollback	
6		*Reads x=100*
7		x 🡐 x – 1
8		Write (x)
9		Commit

The transaction T2 is stopped at moment 4. After T1 rolls back, T2 reads the initial value of x. Note that if T1 were committed instead of rolling back, T2 would read the updated value of x after commitment of T1.

Fuzzy read:

Time	T1	T2
1	Read (x)	
2		Read (x)
3	…	x 🡐 x – 1
4		Write (x) *Waits because T1 holds the shared lock (read at moment 1)*
5	Read (x)	*Waits*
6	Commit	
7		Commit

Transaction T2 is stopped by the lock, and T1 continues with repeatable reads.

Phantoms:

Time	T1	T2
1	Read all x_i for which $F(x_i)$ = TRUE *100 rows read*	
2	Update all x_i *100 rows updated*	Read (x_j) (x_j is one of x_i)
3		Delete (x_j) *Waits because T1 holds a shared lock (read at moment 1)*
4	Write all x_i for which $F(x_i)$ = TRUE *Waits because T2 holds a shared lock (read at moment 2)*	
5	*Deadlock: one of transactions has to be canceled.*	
6	*Transaction is interrupted, the write operation is cancelled.*	
7	Rollback	
8		*Delete is performed*
9		Commit

The transactions are in deadlock. The transaction T1 is cancelled and the phantom phenomenon is prevented. Note that if T2 were inserting a row instead of deleting it, then the phantom effect would depend on the granularity of locking which will be discussed next.

Granularity of Locking

In our discussion, we were applying the operations to data items, but never discussed what the data item was. The data item used for concurrency management defines the *Granularity of locking*. The data item can be (from the larger to finer grain):

- Whole database
- Table
- Data block
- Row
- Attribute in a row.

The granularity of locking determines concurrent performance in the database: the finer the grain, the higher the level of concurrency. DBMSs usually lock data blocks or rows. Oracle locks rows of a table. Later in this chapter, we will show how Oracle applies locking of a larger grain in some special situations.

For the above example of the phantom, if the grain is smaller than a table, the insertion is allowed and the phantom phenomenon happens.

Locking in Oracle

Here are some examples on the previously used table test that demonstrate how Oracle applies locking:

Time	T1	T2
1	`SELECT * FROM test;` *100 rows returned*	
2		`SELECT * FROM test;` *100 rows returned*
3		`DELETE FROM test WHERE f1 =1;` *1 row deleted*
4	`UPDATE test SET f1 = 110 WHERE f1 =10;` *1 row updated*	
5		`UPDATE test SET f1 = 120 WHERE f1 =20;` *1 row updated*
6	`UPDATE test SET f1 = 111 WHERE f1 = 20;` *Waits*	
7		`COMMIT;`
8	*0 rows updated*	

We can make the following conclusions about locking in Oracle:

- Read operations share locks: T1 reads rows of the test table at moment 1, and then T2 reads the same rows at moment 2.

- The granularity of locking is a row: T1 updates a row at moment 4, and then T2 updates another row at moment 5 (please note that this is not totally conclusive as this could happen if Oracle locked data blocks and the involved rows were in different blocks; but Oracle does lock rows).
- Write operations apply exclusive locks: T2 applies an exclusive lock to the row at moment 5, then T1 wants to write the same row, but has to wait for the lock to be released.
- Once again we can see that Oracle has the READ COMMITTED isolation level: T1 does not find any rows for its last update because the row with f1 = 20 was updated and committed to f1 = 120 by T2.
- And the last interesting conclusion: Oracle does not apply any reading lock! At moment 3, T2 deleted a row. If Oracle applied a shared lock on rows of the table, then T2 would not be able to apply the exclusive lock to these rows (could not delete the rows) after T1 had read them.

Timestamping

Unlike locking approaches, timestamping does not maintain the serializability of schedules by exclusion. It tries to build a serializable schedule by specific ordering of the operations of transactions.

To order operations, the transaction manager assigns to every transaction T a timestamp that we will denote as ts(T). Timestamps must be unique and monotonically increasing. If ts(Ti) < ts(Tj), we say that Ti is older than Tj (i.e. Tj is younger than Ti).

The Basic Timestamping Algorithm

The main idea of timestamping is that for each pair of conflicting operations, the operation of the older transaction must be executed before the operation of the younger transactions. If O_{1i} and O_{2j} are two conflicting operations from transactions T1 and T2, respectively, and ts(T1) < ts(T2), then O_{1i} has to be executed before O_{2j}.

If the transaction manager receives two sets of operations $\{O_{11}, O_{12}, \dots, O_{1n}\}$ and $\{O_{21}, O_{22}, \dots, O_{2m}\}$, it can build a serializable schedule. For example, if O_{1i} and O_{1k} are conflicting with O_{2j} and ts(T1) < ts(T2), then O_{1i} and O_{1k} will be placed before O_{2j} in the schedule.

Because not all the operations of transactions are available when their concurrent execution starts, the operations are scheduled when they are ready to be executed: the transaction manager checks a new operation against conflicting operations that have been already scheduled. If the new operation belongs to a transaction that is younger than the transaction with the conflicting operations, then the operation in the younger transaction is scheduled for execution. Otherwise, it would be impossible to put the operation in the schedule without violating the rules of timestamping causing the operation to be rejected, and causing the entire transaction to roll back (remember the atomicity property of transactions).

Compare the following execution of transactions to the locking scenario of the sections explaining the types of locks:

Time	T1	T2
1	Read (x)	
2		Read (x) *Is allowed because there is no conflict.*
3	Write (x) *Is not allowed because the conflicting operation was performed by the younger*	

	transaction.	

Time	T1	T2
1	Write (x)	
2		Read (x) *Is allowed because the conflicting operation of the older transaction was executed before.*

In the following example (because we will refer to it several times, we label it as the Timestamping Example) x = 10 and y = 10. For T1 → T2, which means that T2 is younger than T1 and ts(T1) < ts(T2), the result of a serial execution is x = 12 and y = 22.

Time	T1	T2
1	Read(x)	
2	x ← x + 1	
3	Write(x)	
4		Read(x)
5		x ← x + 1
6		Write(x)
7		Read(y)
8		y ← y * 2
9		Write (y)
10	Read (y) *Is cancelled because the younger T2 wrote to y*	
11	y ← y +1 *Is not executed*	
12	Write(y) *Is not executed*	Commit
13	Rollback	

The read operation of the transaction T1 at moment 10 was cancelled because a younger transaction already wrote to y, and all previously executed operations of T1 have to be rolled back. T1 may be restarted with a different timestamp. If the Read(x) operation of T1 were allowed at moment 10, then the result of the execution of the transactions would be incorrect: x = 12, y = 21.

In the case that these transactions were submitting their operations as in the following scenario, then they both would be executed.

Time	T1	T2
1	Read(x)	
2	x ← x + 1	
3	Write(x)	
4	Read (y)	Read(x)
5	y ← y +1	x ← x + 1

6	Write(y)	Write(x)
7	Commit	Read(y)
8		y ← y * 2
9		Write (y)
10		Commit

Note that this scenario will be impossible under the locking approach. For the timestamping approach, when operations of the concurrent transactions come to the transaction manager one at a time, the manager has to be able to recognize a conflict. For example, in the first – unsuccessful – scenario of the Timestamping example, at moment 10 the scheduler has to know that the data item y has been updated by a younger transaction.

To maintain timestamp ordering, the system assigns timestamps to data items. A data item receives read and write timestamps (we will define them with the help of RTS and WTS, respectively). The timestamp of an item is the largest timestamp of all transactions that read or write this data item. Comparing the timestamp of an item with the timestamp of the transaction, the transaction manager can detect the situation when a younger transaction has reached the item before the older one.

The rules of scheduling the operations are the following.

For a transaction T that issues *Read(x)*:

- If ts(T) < WTS(x), then T is aborted. In other words, if a younger transaction has already updated the item, the transaction T cannot read it.

- Otherwise T continues. The read timestamp of an item is reassigned to RTS(x) = max(ts(T), RTS(x)) – either the read timestamp of T or the timestamp of a younger transaction that has read x.

For a transaction T that issues *Write(x)*:

- If ts(T) < RTS(x) or ts(T) < WTS(x), then T is aborted. It means that if a younger transaction has read the item and is using it or has updated the item, then the write of the transaction T is not allowed.
- Otherwise T continues. The write timestamp of x is reassigned to WTS(x) = ts(T).

By applying these rules, the timestamping approach always produces serializable schedules. The disadvantage of the ordering approach is the aborting of transactions, which may create significant overhead. However, timestamping is free from deadlocks.

One more thing has to be said about timestamping. In the second scenario of the Timestamping example, which was serializable, both transactions were successfully completed. What if for some reason transaction T1 is rolled back? This will create a dirty read situation. Timestamping does not prevent dirty reads. That is why timestamping usually is used either in combination with some kind of locking or finalizing the transactions is postponed – the transaction manager delays the execution of the conflicting operation until it receives the notification of how the older conflicting transaction ended.

Let us see how timestamping prevents the phenomena (in all examples below, assume that T1 is older than T2):

Lost Update:

Time	**T1**	**T2**

1	Read (x)	
2	x ← x + 1	Read (x)
3	Write (x) *Is aborted because RTS(x) > ts(T1)*	x ← x – 1
4	Rollback	Write (x)
5	*T1 has to be restarted*	Commit

The write operation of the transaction T1 is aborted and T1 has to roll back. T2 ix successfully completed and there is no lost update.

Dirty Read:

Time	T1	T2
1	Read (x)	
2	x ← x + 1	
3	Write (x)	
4	…	Read (x) *Is allowed because ts(T2) > WTS (x)*
5	Rollback	x ← x – 1
6		Write (x) *Dirty read*
7		Commit

As was mentioned, timestamping does not protect from dirty reads.

Fuzzy read

Time	T1	T2
1	Read (x)	
2		Read (x)
3	…	x ← x – 1
4		Write (x) *Is allowed because ts(T2) > RTS(x) and ts(T2) > WTS(x)*[1]
5	Read (x) *Is aborted because WTS(x) > ts(T1)*	…
6	Rollback	Commit
7	*T1 has to be restarted.*	

Transaction T1 has to be restarted, and if the next time it does not experience concurrent conflicting transactions, it would have repeatable reads (but the value of x would be different, of course).

Phantoms

[1] WTS (x) is less than ts(T2) because we assume that the last writing to x was performed before T1 and T2.

Time	T1	T2
1	Read all x_i for which $F(x_i)$ = TRUE *100 rows read*	
2	Update all x_i	Read (x_j)
3		Delete (x_j) *Is allowed because ts(T2) > RTS(x).*
4	Write all x_i for which $F(x_i)$ = TRUE *Is aborted because ts(T1) < RTS(x)*	
5	Rollback	Commit
	T1 has to be restarted	

The phantom phenomenon was prevented; the transaction T1 has to be restarted.

The Multiversioning Algorithm

To reduce the overhead of restarting transactions, often systems use a special modification of timestamping – multiversioning. Under this approach, write operations do not modify the database directly. Instead, each update creates a new version of the data item. Each version of the data item is marked by the timestamp of the transaction that created it. The existence of versions is transparent to users. Users refer to an item without knowing with which version of it they are working – the appropriate version is chosen by the system.

The rules of multiversioning are the following:

For a transaction T that issues *Write(x)*:

- If the version x_i that has the largest write timestamp $WTS(x_i)$ satisfies $WTS(x_i) <= ts(T)$ and $ts(T) < RTS(x_i)$, then T is aborted. In other words, if a younger transaction has already read the data item (the version of the item which T has to write), the transaction cannot write the item.
- Otherwise, the transaction T continues and creates a new version x_j with $WTS(x_j) = ts(T)$.

For a transaction T that issues *Read(x):*

- T reads the version x_i of x, where $WTS(x_i)$ is the largest write timestamp less than or equal to ts(T).

The last rule makes multiversioning different from basic timestamping – theoretically, reads never fail. In practice, however, some reads may fail because the storage space for multiple versions is limited – see the explanation below.

Figure 4 shows several versions of data items x, y, and z with their timestamps. The timestamp of a version is the timestamp of the transaction which created this version. From the figure we can say that the earliest versions of x (timestamp 90) was x = 10. Then some transaction with the timestamp 95 updated x and created another version x = 11. After that the transaction with the timestamp 100 again updated x to x = 12. The same transactions updated the data item y. The data item z was updated by older transactions with the timestamps 71 and 86.

Consider three transactions: T1 with ts(T1) = 101, T2 with ts(T2) = 90, and T3 with ts(T3) = 70. Each of them has to read items x and z and then increase them by 1.

T1 is younger than any transaction, which recently accessed items x and z, that is why it will be executed without any problems. T1 will read the latest version of x with the timestamp 100 and then create a new version of x with its own timestamp. It will similarly process the item z.

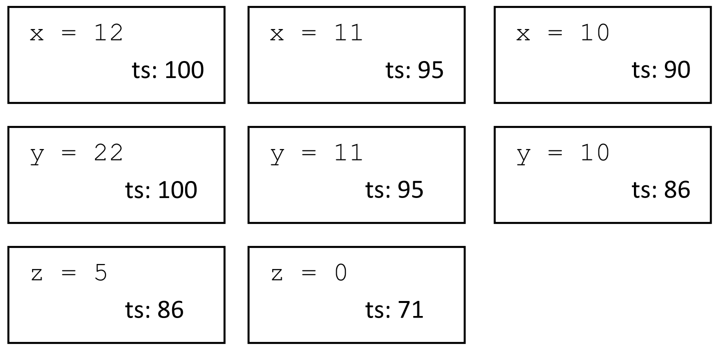


Figure 4. Multiple versions of data.

T2 is able to read from the version of x with the timestamp 90. This is, actually, the version created by T2 itself, as we can see from the timestamp. However, when T2 tries to write the new version of x, the transaction manager discovers that the latest version of x has a higher read timestamp than T2 – 100 – which means that a younger transaction read and wrote the item (actually, the transaction with the timestamp 95 read the version of x available to T2, and after that, the transaction with the timestamp 100 read the version with the timestamp 95), and therefore T2 is aborted.

T3 cannot find appropriate versions of x and z that it can read because its timestamp is smaller than the timestamp of any version of these data items, that is why it has to be aborted as well.

Note that the rules for write operations of the basic time ordering approach do not change. If the discussed transactions want to write, then:

- T1 can write into x, y, and z.
- T2 can write only into z.
- T3 cannot write into any of the three data items.

Versions of data are stored in special memory – *rollback segments*. Rollback segments contain versions of data from committed or still uncommitted recent transactions. Because the size of rollback segments is limited, data versions of new updates overwrite data versions of updates that happened earlier. That is why some long-running transactions, such as T3 in the example above, which started long before other transactions, can fail to find the required version of data and have to be aborted.

Let us analyze multiversioning scenarios for the examples of phenomena (T1 is older than T2):

Lost Update

Time	**T1**	**T2**
1	Read (x)	
2	x 🡐 x + 1	Read (x)

3	Write (x) *Is aborted because ts(T1) < RTS(x)*	x ← x – 1
4	Rollback	Write (x)
5	*T1 has to be restarted*	Commit

The transaction T1 is aborted because a younger transaction already read the data item.

Dirty Read

Time	T1	T2
1	Read (x)	
2	x ← x + 1	
3	Write (x)	
4	…	Read (x) *Is allowed because ts(T2) < WTS (x)*
5	Rollback	x ← x – 1
6		Write (x) *Dirty read*
7		Commit

After T1 rolls back, T2 is successfully completed. However, T2 used dirty data of T1 and wrote the dirty result in the database.

Fuzzy read

Time	T1	T2
1	Read (x)	
2		Read (x)
3	…	x ← x – 1
4		Write (x)
5	Read (x) *Reads the version of x as it was at moment 1*	…
6	Commit	Commit

The transaction T1 does not encounter the fuzzy read because it can only read the version of data which is consistent with its timestamp.

For the phantom example, the processing is performed in the same manner as for the basic timestamping approach.

Under multiversioning, phenomena examples have less aborted transactions than for the basic timestamping: in the fuzzy read example under the timestamping approach one transaction was aborted, while for the multiversioning approach both transactions were processed and produced a consistent database state.

Multiversioning changes one of the scenarios of the concurrent execution of transactions for the case 1 with bank accounts.

Time	Scenario 2	
	John	**Mary**
1	John tries to withdraw $50. The application reads the balance on the account; the balance is $100.	Mary tries to withdraw $75. The application reads the balance on the account; the balance is $100.
2	Money is withdrawn.	
3		Money is withdrawn.
4	The balance is updated to $50 *Is aborted because Mary's transaction has read the account's row at moment 1, RTS(balance) > ts(John's transaction)*	
5	Rollback	The balance is updated to $25.

Preventing inconsistency, multiversioning causes rollback of one of the transaction.

In the multiversioning approach, storage space is traded for performance – though the system has to use more space for storing multiple versions of data, there is less overhead of aborted transactions.

Timestamping in Oracle

The following is another example of the execution of concurrent transactions in Oracle:

Time	T1	T2
1	`SELECT * FROM test;` *returns 100 rows*	
2		`SELECT * FROM test;` *returns 100 rows*
3		`DELETE FROM test WHERE f1=1;` *1 row deleted*
4	`SELECT * FROM test;` *returns 100 rows*	
5	`DELETE FROM test WHERE f1 =2;` *1 row deleted*	
6	`SELECT * FROM test;` *returns 99 rows*	
7		`COMMIT;`
8	`SELECT * FROM test;` *returns 98 rows*	

From this example we can conclude that Oracle uses multiversioning. At moments 4 and 6 the transaction T1 reads the version of data consistent with its own timestamp: at moment 4 it is the same version as in the beginning of the transaction though the transaction T2 has deleted one row, and at moment 6 it is its own new version of data. However, at moment 8, after T2 commits the update of data, the transaction T1 reads the current version of data and not the version consistent with its timestamp. We remember that Oracle uses the READ COMMITTED isolation level, which makes transactions see committed changes of other transactions.

The special behavior of multiversioning in Oracle is also seen at moment 5, when T1 is allowed to write data (delete a row), which was previously read by the younger transaction T2 at moment 2. By the rules of multiversioning, this delete operation had to be aborted. This situation shows that Oracle does not use read timestamps.

Transaction Management in the Distributed Database

Serializability in the Distributed Database

Let us discuss how serializability is supported in the distributed database. First, as the following example shows, the definition of serializability has to be changed for the distributed replicated database.

Consider the data item x = 10 that is replicated on two sites. Two transactions T1 and T2 are executed concurrently.

T1
Read(x)
x ← x+1
Write(x)
Commit

T2
Read(x)
x ← x*2
Write(x)
Commit

Here are the two serial schedules: S1 is generated on the first site, and S2 is generated on the second site:

S1 = {R1(x), U1(x), W1(x), C1, R2(x), U2(x), W2(x), C2}
S2 = {R2(x), U2(x), W2(x), C2, R1(x), U1(x), W1(x), C1}

As you can see, the order of transactions in these schedules is different. After the transactions are finished, on the first site x = 22 (because T1 preceded T2), and on the second site x= 21 (because T2 preceded T1). These states of replicated data violate the mutual consistency of local databases in the replicated database.

Distributed replicated database schedules that maintain mutual consistency are called *one-copy serializable*. A one-copy schedule is serializable if:

- Each local schedule is serializable.
- Each pair of conflicting operations is executed in the same order in all local schedules (in our example, the order of execution of conflicting operations W1 and W2 on the second site was different from the order of these operations on the first site).

Locking Approaches

In the distributed database, locking has to be applied to all replicas of accessed data and to remotely accessed data. Distributed lock management can be performed in several ways:

- *Centralized*. One site is defined as the primary site and it is in charge of locking all replicated or remotely accessed data.
- *Primary copy locking*. One copy of shared data is designed for locking; all other sites must go to this site for permission to access data.
- *Decentralized locking*. All sites share lock management and execution involves coordination of schedulers of all sites.

Timestamping Approaches

Generating timestamps for a distributed database can be implemented with the help of a global monotonically increasing counter (which is difficult to maintain in a distributed environment) or with autonomous counters on each site. With autonomous counters, to preserve the uniqueness of identifiers, the site's number is usually attached to the transaction timestamp.

Transactions and Performance

The design of transactions and making them correspond to business rules is the responsibility of the application programmer. In a multi-user environment, the correctness of transactions requires the appropriate use of concurrency support mechanisms of the particular DBMS. Because the support of consistency of the database comes at the cost of performance, application programmers must carefully use the concurrency support features of the database. The design of transactions can have a significant impact on the performance of database applications.

> **The design of transactions has to be based on the business rules and take into consideration the features of the DBMS and the conditions under which transactions are executed.**

We will discuss some basic recommendations for the design of transactions.

Reduce Locking

Locking reduces concurrency on the database. Most DBMSs provide locking of different granularity. Use locking of the finest granularity allowed by the correctness conditions of your transaction. For example, your transaction has to produce a report based on data from a table, and it is important that the data remain unchanged while the report is being built. The simplest way to do it is to lock the whole table. In many DBMSs this would be unnecessary and inconsiderate to other transactions. Because our transaction only reads data from the table, the better way to preserve the transaction's consistency is to utilize multiversioning and the appropriate isolation level of the DBMS. For example, in Oracle you may define the transaction as READ ONLY (this elevates the isolation level from the default READ COMMITTED), and the transaction will see the version of data consistent with its timestamp.

Make Transactions Short

The length of the transaction affects database performance. The longer a transaction is, the longer other transactions may have to wait for the transaction to finish and release the locks. On the other hand, for a longer transaction it is more probable that it would be locked by other transactions that keep locks.

However, breaking a transaction into several smaller transactions should not endanger the consistency of the database. For example, in *Case 5*, if we consider dividing the money transfer transaction into two: one for withdrawal and another for depositing, then it will compromise the consistency of the database. If a failure happens between these two smaller transactions, then we will not be able to rollback the result of the first transaction and will end up with an inconsistent database.

In a different situation, when a bank transaction includes performing some operation (e.g. depositing money into an account) and then requesting a report on the account's activity, the corresponding database transaction can be broken into two: one for updating the row with the account's data, and another for reading data about the account's activity. In this case, locks applied for updating the account will be released immediately after the update and make the data available for other transactions.

Choose Isolation Level

Most DBMSs support several isolation levels. Higher isolation levels guarantee better consistency, but provide less concurrency than the lower ones. Choose the lowest isolation level without compromising the consistency of your transaction. For example in Oracle, if your transaction performs read operations only, and you want the reads to be consistent with respect to some moment, then there is no need to execute the transaction in SERIALIZABLE mode, it is sufficient to declare the transaction as READ ONLY. Because the READ ONLY isolation level is lower than SERIALIZABLE, your transaction will allow for more activities on the database

Evaluate the Importance of Consistency

In the cases with bank operations, the correctness of transactions is required by business rules. However, there are situations when absolute correctness is desirable but not required. For example, imagine an application for flight reservations. Each reservation transaction includes: 1) requesting the list of available seats, 2) asking a customer about the seat he/she wants, and 3) booking the seat. If all these actions are organized in one transaction, then while a customer is thinking about the seat, all other reservation transactions are locked. If we agree to occasional inconsistencies in the flight reservation processing, we may break the transaction into two by making the request for available seats a separate transaction. The worst thing that can happen is that while a customer is thinking about the seat and decides to book a particular one, another customer takes this seat. In this case, a customer needs to restart the reservation process. This approach, though it causes some inconsistencies in business processing, does not compromise the consistency of data.

Apply Other Measures

In some cases, concurrency issues can be resolved with the help of other measures.

Conflicts for data can be resolved with the help of organizational routines. For example, if a long transaction has to be executed fast, and we do not want other transactions to interfere with its execution and lock the data, instead of locking the whole tables used by this transaction, we may execute this transaction at night, when there is no other updating activity on the database.

The consistency of transactions can be enforced by database design. For example, suppose a long transaction has to perform changes to most of the rows of the table T every day. The changes are based on data from the table S. If we do not organize these updates as one transaction, then in case of failure during the execution of the updates, the table T may become inconsistent – part of the rows that were successfully modified will be as of the current day, while another part will remain as of the previous day. Such a transaction, however, can cause serious performance problems for other transactions because during its execution many rows of the table T will be locked, or it may experience problems itself because there is a possibility that another transaction will try to read or modify data from T. We may choose another approach and add a column, which will keep the date of the last update of a row, to the table T. Now we can organize the update of the table T as a series of transactions, each of which updates one row if the date of the last update of the row is not the current day. If the updating procedure fails, we can restart it as many times as necessary.

We discussed distributed design as an effective way to improve performance. When data is efficiently localized, the possibility of transaction conflicts for data is reduced.

How Oracle Manages Data Concurrency and Consistency

Oracle maintains data consistency in the multi-user environment with the help of combining the multiversioning consistency model, various types of locks and different isolation levels of transactions.

A transaction begins with the first executable SQL statement (except for transactions not using the default isolation level). The transaction ends when any of the following happens:

- COMMIT or ROLLBACK statement.
- DDL statement (such as CREATE, DROP, RENAME, ALTER). The current transaction commits any DML statements and then DDL statement is executed and committed.
- The user disconnects from Oracle (the current transaction is committed).
- The user process terminates abnormally (the current transaction is rolled back).

After a transaction ends, the next executable SQL statement automatically starts the next transaction.

1.1.1 Query Consistency

Oracle *automatically* provides *read* consistency for separate *queries*. This means that the query will see all the data it needs, the way the data was when the query started. This is called *statement-level read consistency*. If necessary, Oracle also provides read consistency of all queries in the transaction – called *transaction-level read consistency*.

To provide these consistent views of data, Oracle uses multiple versions of data maintained in rollback segments. The rollback segment contains older values of data that have been changed by uncommitted or recently committed transactions. Figure 5 shows how Oracle provides statement-level read consistency using data in rollback segments.

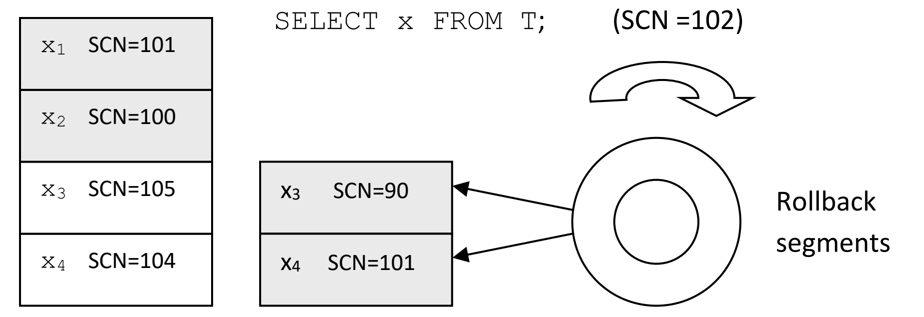


Figure 5. Transactions and Read consistency.

When the query is submitted, it is assigned a timestamp, which in Oracle is called the current system change number (SCN). The query in Figure 5 has SCN = 102 and it will retrieve the versions of the requested rows with SCN less than or equal to its own SCN (highlighted in the figure). The third and fourth rows of the table were modified by younger transactions, and the system retrieves appropriate versions of data from the rollback segment. Therefore, each query sees the latest changes that have been committed before the query began. If other transactions that occur during the query execution perform changes to the requested data, then the versions of data consistent with the SCN of the query are reconstructed from the rollback segments, guaranteeing that consistent data is returned for each query. A query never sees dirty data or any of the changes made by transactions that commit during the query execution.

In rare situations, Oracle cannot return a consistent set of results for a long-running query. This occurs when the versions of data needed by the query are overwritten in the rollback segment by versions of younger updates because of the limited storage in the rollback segments. This can be avoided by creating more or larger rollback segments.

If at any time during execution an SQL statement causes an error, all effects of the statement are rolled back. For example, the query UPDATE T SET x = x*1.1 needs to read consistent versions of x to perform its update. If for some row, an appropriate version of x is unavailable from the rollback segment (e.g. due to the situation described above), the query is aborted and all updates that took place are rolled back.

With the help of the SERIALIZABLE isolation level (see details below) Oracle can enforce transaction-level read consistency. When the transaction is executed in the serializable mode, it sees all data as of the time when it started and the changes performed only by that transaction without seeing the committed changes of other transactions. Transaction-level read consistency produces repeatable reads and does not expose the query to phantoms.

Oracle provides three isolation levels: READ COMMITTED, SERIALIZABLE, and READ ONLY.

READ COMMITTED Isolation Level

This is Oracle's default isolation level. Each query executed by a transaction sees only data that was committed before the query (not the transaction) began. The query will never read dirty (uncommitted) data. Oracle does not prevent other transactions from modifying data read by the transaction because read operations do not apply locks. Therefore, data read by the transaction can be modified by other transactions, and as a result, if the transaction executes a particular read query several times, it may experience both non-repeatable reads and phantoms.

Let us analyze examples of phenomena in Oracle with READ COMMITTED isolation level:

Lost Update

Time	T1	T2
1	`SELECT x FROM T;` *Returns 0*	
2		`SELECT x FROM T;` *Returns 0*
3	`UPDATE T SET x = x + 1;` *Is allowed because the read of T2 did not apply a lock*	
4	`COMMIT;`	
5		`UPDATE T SET x = x - 1;` *Is allowed because T1 committed and released locks*
6		`COMMIT;` *Lost update of T1*

In this particular modification of our initial scenario the lost update phenomenon is not prevented because Oracle SELECT operations do not apply locks (compare with lost update under the theoretical locking approach). Note that the lost update of the first transaction could be prevented by executing it with the

SERIALIZABLE isolation level (see the following section).

Dirty Read

Time	T1	T2
1	`SELECT x FROM T;` *x = 100*	
2	`UPDATE T SET x = x + 1;`	
3		`SELECT x FROM T;` *No dirty reads, because it does not see the uncommitted write of T1; it reads x = 100 from the older version*
4	`ROLLBACK;`	
5		`...`

Oracle lowest isolation level – READ COMMITTED – prevents transactions from seeing dirty data of other transactions.

Fuzzy read

Time	T1	T2
1	`SELECT x FROM T;` *x = 100*	
2		`UPDATE T SET x = x - 1;`
3	`SELECT x FROM T;` *x = 100. No fuzzy because T1 does not see uncommitted changes of T2*	
4		`COMMIT;`
5	`SELECT x FROM T;` *x = 99. Fuzzy read because T1 sees committed changes of T2*	...
6	`COMMIT;`	

The fuzzy read does not happen at moment 3 because of the READ COMMITTED isolation level. However, this isolation level does not prevent the fuzzy read at moment 5, after T2 committed its changes. READ COMMITTED does not prevent phantoms as well.

SERIALIZABLE Isolation Level

The SERIALIZABLE isolation level can be set for the instance, session, or a particular transaction. The serializable transaction sees only those changes that have been committed at the time the transaction began, and the changes made by the transaction itself through INSERT, UPDATE, and DELETE statements. Serializable transactions do not encounter non-repeatable reads or phantoms. In situations when Oracle cannot serialize a transaction due to a committed delete, insert or update of a younger transaction, it generates an error. The following is an example using the table test introduced earlier in this chapter:

Time	T1	T2
1	`SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`	

2	`SELECT f2 FROM test WHERE f1 <10;`	
3	...	`UPDATE test SET f2 = 101 WHERE f1 = 1;`
		`COMMIT;`
4	`UPDATE test SET f2 = 101 WHERE f1 = 1;` *Cannot serialize access*	

Serializability in Oracle is implemented as a complete isolation of the transaction from other transactions. Each serializable transaction is executed as if it were the only transaction. The following is an example that shows that the result of two serializable transactions executed concurrently does not necessarily agree with the result when these transactions are executed serially (from [Kyte 2005]). The two transactions shown perform operations on the empty tables A(x) and B(x):

Time	T1	T2
1	`SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`	
2		`SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`
3	`INSERT INTO A SELECT COUNT(*) FROM B;`	
4		`INSERT INTO B SELECT COUNT(*) FROM A;`
5	`COMMIT;`	
6		`COMMIT;`
7	`SELECT x FROM A;` *x = 0*	
8		`SELECT x FROM B;` *x = 0*

In the serial execution of these transactions, one of them will return x = 1. The concurrent execution of transactions returns a result that is inconsistent with serial execution.

READ ONLY Isolation Level

The read-only transaction cannot include INSERT, UPDATE, and DELETE statements; it sees only those changes that have been committed at the time the transaction began. It behaves as a serializable transaction with only SELECT statements. If read consistency of the transaction is important and the transaction does not contain modifying statements, then it is recommended to use this isolation level. The following is an example of the fuzzy read scenario for a transaction with read-only isolation level:

Time	T1	T2
1	`SET TRANSACTION ISOLATION LEVEL READ ONLY;`	
2	`SELECT x FROM T;` *x = 100*	
3		`UPDATE T SET x = x - 1;`

4	`SELECT x FROM T;` *x = 100. No fuzzy read because T1 does not see uncommitted changes of T2*	
5		`COMMIT;`
6	`SELECT x FROM T;` *x = 100. No fuzzy read because of the read only isolation level.*	…
7	`COMMIT;`	

SELECT FOR UPDATE

Some of the phenomena (e.g. lost update) happen in Oracle as it does not apply read locks by default. However, we can explicitly request a read lock with the statement SELECT … FOR UPDATE. For example, the lost update phenomenon example with SELECT FOR UPDATE will be processed in the following way:

Time	T1	T2
1	`SELECT x FROM T WHERE x = 100 FOR UPDATE;`	
2		`SELECT x FROM T WHERE x = 100 FOR UPDATE;`
3	`UPDATE T SET x = x + 1 WHERE x =100;` *Waits because of the read lock applied by T2*	
4		`UPDATE T SET x = x - 1 WHERE x =100;` *Waits because of the read lock applied by T1*
5	*Deadlock is resolved by aborting the last operation of T1.*	
6	`ROLLBACK;` *T1 has to rollback. The lock of x is released*	
7		*1 row updated*
		`COMMIT;`

Transactions behave similar to the basic locking approach: one of the transactions has to be aborted, and the consistency of the database is preserved.

Locking Data in Oracle

Row-Level Locking

Both read committed and serializable transactions use row-level locking. Oracle uses fine-grain locking to provide a high degree of concurrency. Oracle's row locking is fully automatic and requires no user action. Implicit locking occurs for all modifying statements. The read lock has to be applied manually with the help of the SELECT FOR UPDATE statement. Oracle also allows users to manually apply several table locks, which are discussed later.

The combination of multiversioning and row-level locking produces the following outcomes for concurrent executions of transactions in Oracle:

- Readers of rows do not wait for writers of the same rows.
- Writers of rows do not wait for readers of the same rows (unless SELECT...FOR UPDATE is used).
- Writers only wait for other writers if they attempt to update the same rows.

The transaction acquires the exclusive DML lock for each individual row modified by one of the following statements: INSERT, UPDATE, DELETE, and SELECT with the FOR UPDATE option.

Oracle automatically detects deadlock situations and resolves them by aborting one of the statements involved in the deadlock, in this way releasing one set of the conflicting row locks.

Table Locking

The only DML locks that Oracle acquires automatically are row-level locks. There is no limit to the number of row locks held by a statement or a transaction.

The following simple example shows that row locks are accompanied by table locks:

Time	T1	T2
1	`UPDATE T SET x = 0 WHERE x =100;`	
2		`ALTER TABLE T ADD (y NUMBER);` *Is waiting*
3	`ROLLBACK;`	
4		*Table is altered*
5	`UPDATE T SET x = 0 WHERE x =100;`	
6		`DROP TABLE T NOWAIT;` *Error message*

The transaction T1 locks rows of the table T. When the transaction T2 tries to perform a DDL operation on the table at moment 2, it has to wait because this operation requires locking the whole table exclusively, which is impossible at the time. When the transaction T2 issues a DDL statement at moment 6 with the option NOWAIT (execute immediately if possible), the system returns an error message because the transaction T1 locked some rows of the table and it is impossible to execute the DROP TABLE operation immediately. Row locks are preventing the whole table from being dropped.

Table locks caused by DML statements are imposed on the table transparently to users. Additionally, Oracle allows users to explicitly apply table locks.

Table locks can be held in one of several modes of different restrictiveness starting with the least restrictive: row share (RS), row exclusive (RX), share (S), share row exclusive (SRX), and exclusive (X). A lock mode applied by a transaction defines what operations can be executed on the table by other transactions and in what modes other transactions can lock the table.

The table below shows table lock modes caused by different statements and operations that these locks permit and prohibit.

SQL Statement	**Mode of Table Lock**	**Lock Modes Permitted**				
		Row Share (RS)	**Row Exclusive (RX)**	**Share (S)**	**Share Row Exclusive (SRX)**	**Exclusive (X)**
`SELECT...FROM table...`	none	Y	Y	Y	Y	Y
`INSERT INTO table ...`	RX	Y	Y	N	N	N
`UPDATE table ...`	RX	Y*[2]	Y*	N	N	N
`DELETE FROM table ...`	RX	Y*	Y*	N	N	N
`SELECT ... FROM table FOR UPDATE OF ...`	RS	Y*	Y*	Y*	Y*	N
`LOCK TABLE table IN ROW SHARE MODE`	RS	Y	Y	Y	Y	N
`LOCK TABLE table IN ROW EXCLUSIVE MODE`	RX	Y	Y	N	N	N
`LOCK TABLE table IN SHARE MODE`	S	Y	N	Y	N	N
`LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE`	SRX	Y	N	N	N	N
`LOCK TABLE table IN EXCLUSIVE MODE`	X	N	N	N	N	N

In most cases, database programmers only need to properly define transactions, and Oracle will automatically manage locking. However, overriding default locking can be useful in some situations such as:

- We know that our transaction will lock most of the rows of a table. Then locking the whole table in the beginning of the transaction gives better performance than applying locks to individual rows.
- The business logic of the transaction prohibits any phenomena, and the transaction is important and cannot be aborted. We had an example showing that the serializable isolation level, though it prevents inconsistencies, can cause an abortion of the transaction. Using exclusive table locks that do not allow other transactions any updating activity on the table (e.g. X or SRX) will protect the transaction from the phenomena.
- The transaction has to have exclusive access to the table to prevent other transactions from any modifying actions that can cause delays in the transaction execution.

An interesting illustration of Oracle locking is its handling of referential integrity. For the Manufacturing

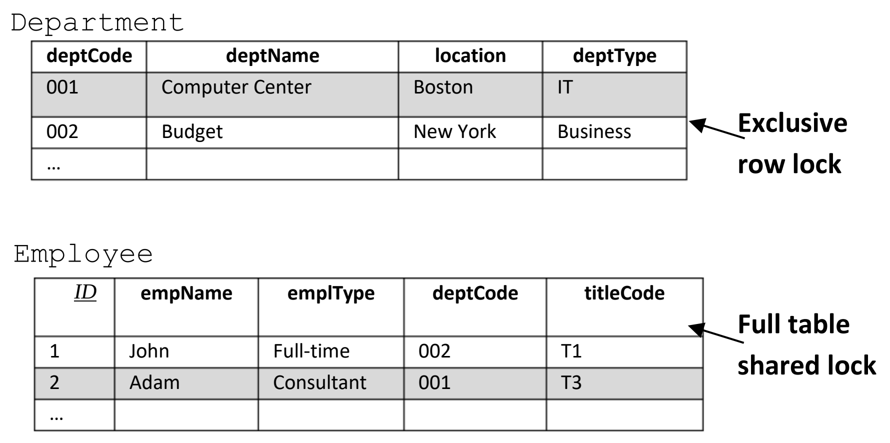
[2] Y* means that only the rows affected by the operation are locked.

Company case let us consider the situation when for one of the rows of the table Department the value of the primary key is modified or the row is deleted. The problem we can run into here is violation of the foreign key constraint when a concurrent transaction tries to use the old or new value of the deptCode in one of the rows of the child table Employee.

Figure 6 illustrates the locking on the parent and child tables when a user attempts to change the value of the deptCode (for the deptCode = '001').

The following steps take place:

- The updated row is locked exclusively in a regular manner.
- The database acquires a *full table shared lock* on all rows of the table Employee during the primary key modification of department '001'.
- This lock enables other sessions to query but not modify the table Employee. For example, neither attributes of the table can be updated nor a new record can be inserted. The table lock on Employee releases *immediately* after the primary key modification on the departments table completes (note that in Oracle 8 the lock was released at the end of the transaction). If multiple rows in departments undergo primary key modifications, then a table lock on employees is obtained and released once for each row that is modified in departments.



Department

deptCode	deptName	location	deptType
001	Computer Center	Boston	IT
002	Budget	New York	Business
…			

Employee

ID	empName	emplType	deptCode	titleCode
1	John	Full-time	002	T1
2	Adam	Consultant	001	T3
…				

Figure 6. Locking in support of referential integrity.

Let us analyze what happens in different situations:

- Update of deptCode = '001'or delete of this row. Because there are dependent rows in the child table, the update or delete will not happen.
- Update of deptCode or delete of the row that are not referenced in the child table. Updates or inserts on the child table that might reference the old values of the deptCode wait for a row-lock on the parent table to clear.
- Insert of a new row in the parent table. Updates or inserts on the child table that might reference the new value of the deptCode will not happen until the first transaction on the parent table commits.

In Appendix 5 there is an example of the concurrent behavior of two transactions that apply different types of locks.

Preventing Phenomena in Oracle Transactions

When we design transactions for database applications, we pursue the following goals:

- Maintain consistency of the business data or, in other words, avoid phenomena
- Minimize the chance of the transaction of being delayed or interrupted
- Minimize imposition on other transactions.

For example, to avoid the lost update illustrated in several scenarios, we can explicitly apply shared row lock (`FOR UPDATE`), or lock the table in the `ROW SHARED` lock, or execute the transaction with the `SERIALIZABLE` isolation level. The first approach seems to be most appropriate as it is less imposing on other transactions than the second approach and does not create potential problems for the transaction itself when it can be interrupted by younger transactions.

If we need to prevent fuzzy read, then the most appropriate would be to use the READ ONLY isolation level: we prevent fuzzy reads without preventing other transactions from reading or modifying data we deal with. If our transaction has modifying operations, then we can lock data (rows or the whole table) in the SHARED lock. Using SERIALIZABLE isolation level has the same shortcomings as in the case of the lost update.

Examples: Implementation of Transactions and the Design of the Database for Performance and Consistency

Transactions in Applications

Earlier in this chapter, we discussed examples of interactive transactions, where a user submitted a transaction consisting of several operations, waited for the response of the database, and then continued executing the transaction depending on the results of the previous operations. Recall, for example, transferring money from one account to another. A user withdrew money from one account, and then deposited the same amount to another account. If both operations were correct, the user committed the actions. If at least one of the operations was aborted, then executed operations were rolled back. Transactions in applications are implemented as sequences of commands, and when a transaction is started, the system tries to execute all the commands one after another. The logic of the transaction and different actions that have to be performed depending on the results of the operations need to be programmed. The following is an example of a bank money transfer transaction implemented with the help of an Oracle stored procedure:

```
CREATE OR REPLACE PROCEDURE transfer
       (par_account1 NUMBER, par_account2 NUMBER, par_amount NUMBER)
AS
   nAmount NUMBER;
BEGIN
   -- Check whether the amount can be withdrawn
   SELECT Balance INTO nAmount FROM Accounts WHERE account = par_account1;

IF nAmount >= par_Amount THEN
       UPDATE Accounts
       SET Balance = Balance - par_amount
       WHERE account = par_account1;
```

```
        UPDATE Accounts
        SET Balance = Balance + par_amount
        WHERE account = par_account2;
END IF;
-- If we are here, then all actions were successfully executed
-- and must become permanent.
COMMIT;

EXCEPTION
   WHEN OTHERS THEN
    -- If we are here, then something happened
    -- during execution of the procedure. Any actions that were
    -- executed should be undone.
        ROLLBACK;
END;
```

If anything happens during the execution of the procedure, e.g., one of the update statements is aborted because of a deadlock, and Oracle raises an exception and takes the procedure to the exception handling section. In our procedure, this section of the procedure ensures that any performed actions are undone by issuing ROLLBACK. In case all actions were successful, the procedure gets to the end, where the actions are made permanent with the help of COMMIT. A bank application contains the call for the execution of the procedure. For example, for transferring $100 from the account 1234 to the account 5678, the application will execute the call:

```
EXEC transfer (1234, 5678, 100);
```

Though this transaction prevents inconsistency in case of failure or possible errors, it does not consider problems of concurrent access to data. Because Oracle does not apply read locks, after the check of the balance on the first account, another transaction may change the value of the balance and cause data inconsistency, for example, as in the *Case 1*. To guarantee data consistency, the statement with the check of the balance has to be rewritten in the following way:

-- *Check whether the amount can be withdrawn*

-- *Apply FOR UPDATE option to prevent balance from being updated by another user*

SELECT Balance INTO nAmount FROM Accounts WHERE account = par_account1
FOR UPDATE;

With the help of this simple example, we once again want to remind readers that the database application and the transactions included in it have to be implemented with a deep understanding of the features of the DBMS and knowledge of how the database is organized.

This transaction can be implemented in the application in any application language, e.g. in Java. The logic of the Java program will be similar to the logic of the Oracle procedure: it will contain read and write accesses to the database, analyses of the results of each operation and corresponding actions on finishing the transaction either by rollback or commit.

Concurrency and the Design of Tables

The way the DBMS provides data processing can cause performance problems in the case of concurrent transactions. Chapter 2 explains the basics of data storage, packing of data, and how the system uses storage parameters to manipulate the data. For example, if a row is to be inserted into a table, the system checks the list of available blocks for the table (called FREELIST) and inserts the row in a block from the list. If after doing this operation the block becomes full (no more rows can be inserted), the system deletes the block from the FREELIST. When data are deleted from the table, the FREELIST is also involved – some blocks may be returned to it depending on the setting of the parameter PCTUSED. Therefore, when processing data from a table, the system performs manipulations on the table's FREELIST, and to maintain consistency of the FREELIST in the case of concurrent actions, it has to apply locks on the FREELIST, as with any other resource. For intensive concurrent inserting and deleting activities on the table, performance can degrade because concurrent actions will be waiting for each other to release the locks of the FREELIST. That is why for the tables with expected concurrent insert and delete actions it is recommended to increase the number of free lists of the table. With multiple lists, more operations, each working with a separate list, can be executed concurrently. The number of free lists of a table is defined in the table's definition:

```
CREATE TABLE test (. . .) STORAGE (FREELISTS 3);
```

Summary

Consistency of the database can be violated by concurrent execution of database operations or by failure of the system. The concept of a transaction as a logical unit of work is introduced to prevent incorrect states of the database. The transaction is a tool of consistent and reliable database processing. Four properties define transaction management: atomicity, consistency, isolation, and durability. Consistency of a transaction in the multi-user environment depends on the isolation property – how well the transaction is shielded from the interference of other transactions. ANSI defines four levels of isolation: READ UNCOMMITTED, READ COMITTED, REPEATABLE READ, and SERIALIZABLE. With different levels of isolation, concurrent transactions can experience such phenomena as lost updates, dirty reads, fuzzy reads, and phantoms. The Serializable level guarantees excluding all phenomena.

The goal of transaction management is to produce serializable schedules of execution of concurrent transactions. Serializable schedules result in the same database state as when transactions are executed serially, and they combine the correctness of serial execution with the better performance of concurrent execution. The transaction manager maintains serializable schedules with the help of locking and timestamping mechanisms.

The transaction applies locking to prevent concurrent transactions from accessing the same data.. Different DBMS apply locking of different granularity, from the row to the whole database. Granularity of locking defines the level of concurrency – the smaller the grain, the higher the level of concurrency. The disadvantage of locking is the possibility of a deadlock. Deadlocks are resolved by the transaction manager.

Under the timestamping approach, operations of concurrent transactions are executed in an ordered, serializable manner. If the transaction manager cannot generate a serializable schedule for the transaction, then the transaction is aborted. Multiversioning is a modified version of timestamping, which allows reducing the number of aborted transactions with the help of storing multiple versions of data.

The design of transactions for database applications plays an important role in database performance. General recommendations for transaction design include using fine grain locking, lowering the isolation level, and splitting long transactions into shorter ones.

Oracle transaction management is based on combinations of locking, multiversioning, and different isolation levels.

Review Questions

1. How can concurrent operations interfere with each other and cause incorrect database states?
2. Why can a system failure cause data inconsistency? Give an example of inconsistent database computing caused by failure.
3. What is a transaction?
4. How is a transaction defined by its properties?
5. What inconsistencies can occur during concurrent database access? Describe situations with different phenomena.
6. What levels of transaction isolation are defined by ANSI? What phenomena can happen for different isolation levels?
7. What types of access to data are conflicting?
8. What types of locks are applied by the database?
9. Can two transactions share a lock? What type of lock is it?
10. What is a deadlock? Give an example of a deadlock between two transactions.
11. What happens if data needed by a transaction are locked by another transaction?
12. What is the goal of the timestamping approach? Define the rules of read and write operations for timestamping approach.
13. Is it possible that a transaction cannot be executed in a serializable manner under the timestamping approach? When does it happen, and how does the transaction manager resolves the situation?
14. What is multiversioning? Explain how multiversioning reduces the number of aborted transactions.
15. What is limiting the possibilities of multiversioning?
16. Which approach is more pessimistic, locking or timestamping?
17. How does the isolation level of the transaction influence performance?
18. What are the special features of transaction management in Oracle?
19. What isolation levels can you define for transactions in Oracle?
20. How are read locks implemented in Oracle?
21. When can manual table locks be useful? What are the disadvantages of table locks?

Practical Assignments

The assignments are defined on the Manufacturing Company case. Implement the assignments in Oracle or another DBMS.

1. You need to increase salaries for all titles by 2%. How should you implement this operation in Oracle to prevent inconsistency caused by interruptions of the operation?

2. Build a transaction for transferring an employee from one department to another in:
    a. A centralized database.
    b. A distributed database for the example presented in Chapter 3.

   Explain your transactions and show that they preserve business consistency of the database.

3. Assume that deptCode of the table Employee is not defined as the foreign key to the table Department. Build two transactions in Oracle: one for deleting an existing department, another – for adding a new employee to this department. Implement referential integrity checks in these transactions. Will your transactions guarantee referential integrity? What changes you will make to your transactions to ensure referential integrity?

4. In the example of the distributed database from Chapter 3, distributed referential integrity is supported with the help of a trigger. Explain how referential integrity can be violated by concurrent actions. Show what changes you can make to the trigger to preserve referential integrity in the concurrent environment.

5. An important transaction repeatedly reads data from the table Employee. How will you protect read consistency of this transaction? Show how you can prevent this transaction from being locked by other transactions.

6. An important transaction updates data in most of the rows of the table Employee. How can you protect the consistency of this transaction? Show how you can prevent this transaction from being locked by other transactions.

7. Execute concurrently a pair of transactions T1: x ← y and T2: y ← x for arbitrary values of x and y. Compare to the serial execution of these transactions and explain the result. Suggest how to change the transactions to ensure consistency of their results.

8. A bank database contains the table Balance(acctNumb, Balance) with data about accounts and balances on them, and the table BankOperation(acctNumb, operationDate, operationNumb, amount)with data about operations on accounts, which includes operation date and number and the amount of money involved in the operation (positive for deposits and negative for withdrawals). At the end of the business day, the table Balance is updated by data from the table BankOperations. Implement this updating procedure in Oracle and demonstrate that the procedure will preserve consistency of data in the case of concurrent access or failure.

9. For each of the following situations, describe what Oracle transaction management tools you would use and explain your choice:
    a. A report with calculations on data from multiple tables.
    b. A report based on several records of one table.
    c. Analysis and processing of all records of a table.
    d. Analysis and processing of data from several tables.