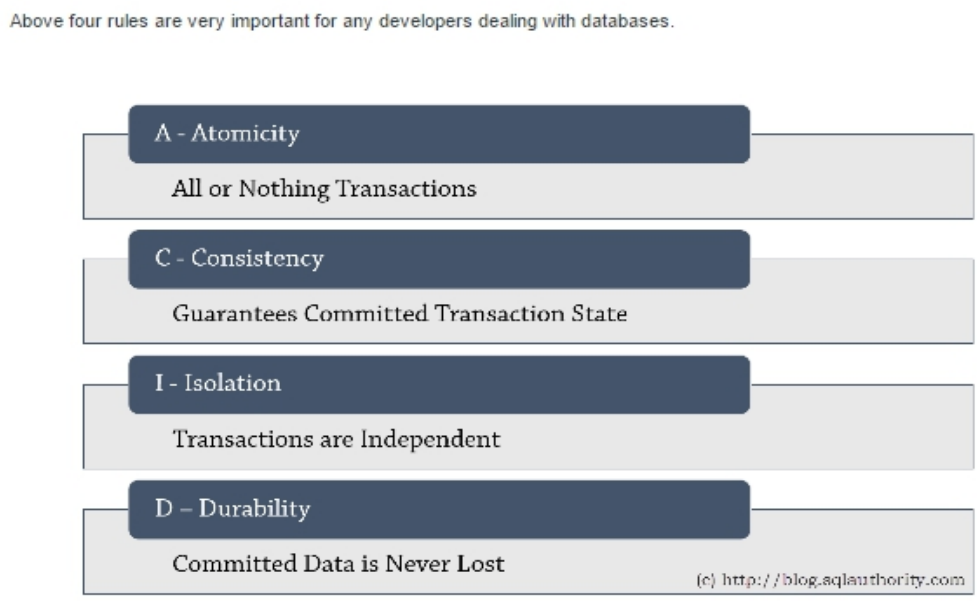
Transaction Isolation

ACID: Transaction property.



A. Transaction Isolation Levels

1. READ UNCOMMITTED

Step	Session 1 (Query Window 1)	Session 2 (Query Window 2)

1	`USE AdventureWorks`
	`Go`
	
	`SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED`
	`Go`
	
	`BEGIN TRANSACTION`
	
	`SELECT EmployeeID, ManagerID`
	`FROM HumanResources.Employee`
	`WHERE EmployeeID = 1`
	
	Results | Messages
	EmployeeID ManagerID
	1 1 16
	
	`/*Note that the ManagerID is 16 for EmployeeID 1. */`	

2		`USE AdventureWorks`
		`Go`
		
		`BEGIN TRANSACTION`
		
		`UPDATE HumanResources.Employee`
		`SET ManagerID = 12`
		`WHERE EmployeeID = 1`
		
		`SELECT EmployeeID, ManagerID`
		`FROM HumanResources.Employee`
		`WHERE EmployeeID = 1`
		
		EmployeeID ManagerID
		1 1 12
		
		`/*Note that the ManagerID has been updated to 12, and that the transaction has not yet committed.`
		
		`(Also note for later that this isolation level allows a transaction to update data that have been read by another transaction. The transaction in Session 1 read the row of EmployeeID 1 and we just changed that row here in the transaction in Session 2.) */`

3	`SELECT EmployeeID, ManagerID`
	`FROM HumanResources.Employee`
	`WHERE EmployeeID = 1`
	EmployeeID ManagerID
	1 1 12
	
	`/*Note that the update made in Session 2 is visible in Session 1 even though it has not yet been committed in Session 2! This is known as a dirty read because the data we read is not yet "real."`
	`*/`
	
	`ROLLBACK TRANSACTION`	

4		`ROLLBACK TRANSACTION` `/*` `REAL WORLD SCENARIO: (See the ATM example in the blog post.) Your family member deposits $1,000 into your mutual account. Seeing that balance, you decide to increase the amount of your transfer, but your family member's transaction is rolled back and you end up putting the account into a negative balance.` `*/`

2. READ COMMITTED (Default)

Step	Session 1 (Query Window 1)	Session 2 (Query Window 2)

1	`USE AdventureWorks`
	`Go`
	
	`SET TRANSACTION ISOLATION LEVEL READ COMMITTED`
	`Go`
	
	`BEGIN TRANSACTION`
	
	`SELECT EmployeeID, ManagerID`
	`FROM HumanResources.Employee`
	`WHERE EmployeeID = 1`
	
	Results | Messages
	EmployeeID ManagerID
	1 1 16
	
	`/*Note that the ManagerID is 16 for EmployeeID 1. */`	

2		`USE AdventureWorks` `Go` `BEGIN TRANSACTION` `UPDATE HumanResources.Employee` `SET ManagerID = 12` `WHERE EmployeeID = 1` `SELECT EmployeeID, ManagerID` `FROM HumanResources.Employee` `WHERE EmployeeID = 1` EmployeeID ManagerID 1 1 12 `/*Note that the ManagerID has been updated to 12, and that the transaction has not yet committed.` `REAL WORLD SCENARIO: Two passengers on the same flight want to reserve the same seat. The first passenger pulls up the seating chart (this is a read of the data). As she thinks about which seat to select, the second passenger also pulls up the seating chart and decides to go ahead and select seat 21C. The first passenger eventually decides to select 21C, also. When they arrive on their flight, they both claim the same seat. A flight attendant checks the seating chart and sees that the seat is registered to the second passenger. The update to the seating chart made by the first passenger was overwritten by the update made by the second passenger. This is known as a "lost update."`  `)*/`

3	`SELECT EmployeeID, ManagerID`
	`FROM HumanResources.Employee`
	`WHERE EmployeeID = 1`
	
	Results | Messages
	
	`/* Note that the transaction in Session 1 is not allowed to read data that are in the process of updating (updated but not yet committed in Session 2). */`	

4		`ROLLBACK TRANSACTION`
5	`SELECT EmployeeID, ManagerID` `FROM HumanResources.Employee` `WHERE EmployeeID = 1` Results | Messages EmployeeID ManagerID 1 1 16 `/*Note that the transaction in Session 1 is now allowed to read data released by the transaction in Session 2 that has completed. (In this case, the transaction in Session 2 was rolled back, so we see the data as they were before that transaction started (ManagerID = 16). If the transaction in Session 2 had been committed, the ManagerID for EmployeeID 1 would have shown up as 12 here, instead.)` `REAL WORLD SCENARIO: (See ATM example in the blog post.) Your family member deposits $1,000 into your mutual account. When you attempt to access the same account before your family member's transaction completes, you are asked to wait or try again at a later time. */`  `ROLLBACK TRANSACTION`	

3. REPEATABLE READ

Step	Session 1 (Query Window 1)	Session 2 (Query Window 2)

1	`USE AdventureWorks`
	`Go`
	
	`SET TRANSACTION ISOLATION LEVEL`
	`REPEATABLE READ`
	`Go`
	
	`BEGIN TRANSACTION`
	
	`SELECT EmployeeID, ManagerID`
	`FROM HumanResources.Employee`
	`WHERE EmployeeID = 1`
	
	Results | Messages
	EmployeeID ManagerID
	1 1 16
	
	`/* Note that we are reading the row of EmployeeID 1. */`	

2		`USE AdventureWorks`
		`Go`
		
		`BEGIN TRANSACTION`
		
		`UPDATE HumanResources.Employee`
		`SET ManagerID = 12`
		`WHERE EmployeeID = 1`
		
		Results | Messages
		
		`/* Note that the transaction in Session 2 is not allowed to update data that have been read by the transaction in Session 1. This is because Repeatable Read Isolation Level makes sure that all reads of unchanged data are repeatable; in other words, reads of the same data produce the same values.`
		
		`REAL WORLD SCENARIO: (See Step 2 in the previous example.) When the second passenger requests the seating chart, he is told to wait or try again at a later time.`
		
		`*/`

3	`ROLLBACK TRANSACTION`	

4		`SELECT EmployeeID, ManagerID`
		`FROM HumanResources.Employee`
		`WHERE EmployeeID = 1`
		
		EmployeeID ManagerID
		1 1 12
		
		`/* Note that the update can go forward now that the transaction in Session 1 has completed. */`
		
		`ROLLBACK TRANSACTION`

4. SERIALIZABLE

Step	Session 1 (Query Window 1)	Session 2 (Query Window 2)
1	USE AdventureWorks Go SET TRANSACTION ISOLATION LEVEL REPEATABLE READ Go /* Note that we start with the Repeatable Read transaction isolation level. */ BEGIN TRANSACTION SELECT * FROM Production.Location Results Messages LocationID Name CostRate Availability ModifiedDate 1 1 Tool Crib 0.00 0.00 1998-06-01 00:00:00.000 2 2 Sheet Metal Racks 0.00 0.00 1998-06-01 00:00:00.000 3 3 Paint Shop 0.00 0.00 1998-06-01 00:00:00.000 4 4 Paint Storage 0.00 0.00 1998-06-01 00:00:00.000 5 5 Metal Storage 0.00 0.00 1998-06-01 00:00:00.000 6 6 Miscellaneous Storage 0.00 0.00 1998-06-01 00:00:00.000 7 7 Finished Goods Storage 0.00 0.00 1998-06-01 00:00:00.000 8 10 Frame Forming 22.50 96.00 1998-06-01 00:00:00.000 9 20 Frame Welding 25.00 108.00 1998-06-01 00:00:00.000 10 30 Debur and Polish 14.50 120.00 1998-06-01 00:00:00.000 11 40 Paint 15.75 120.00 1998-06-01 00:00:00.000 12 45 Specialized Paint 18.00 80.00 1998-06-01 00:00:00.000 13 50 Subassembly 12.25 120.00 1998-06-01 00:00:00.000 14 60 Final Assembly 12.25 120.00 1998-06-01 00:00:00.000 /* Note that we read a range of data with this SELECT statement.	
2		USE AdventureWorks Go BEGIN TRANSACTION UPDATE Production.Location SET CostRate = 2.0 WHERE LocationID = 2 /* Note that the Repeatable Read Transaction Isolation Level does not allow any updates on a row that has been read, even if that row is part of a range. */ INSERT Production.Location (Name, Availability, CostRate, ModifiedDate) VALUES ('NewLoc1', 1.5, $5.00, GETDATE()) /* Note that even though this insert would change the results of the SELECT in Session 1, it is allowed by the Repeatable Read Transaction Isolation Level. While the Repeatable Read Transaction Isolation Level does not allow updates to a row that has been read, it does not prevent inserts into a range that has been read, effectively allowing non-repeatable reads in that range. REAL WORLD SCENARIO: A transaction needs to calculate the total value of inventory in one step and then use that total in another step to calculate the percentages of that total that each inventory category represents. After all of inventory is read in the first step, a new inventory item is inserted into one of the categories by another transaction before the second step can be started in the first transaction. The percentages calculated in the second step will now be incorrect. */ ROLLBACK TRANSACTION
3	ROLLBACK TRANSACTION SET TRANSACTION ISOLATION LEVEL SERIALIZABLE /* Note that we now switch to the Serializable Transaction Isolation Level. */ BEGIN TRANSACTION SELECT * FROM Production.Location Results Messages LocationID Name CostRate Availability ModifiedDate 1 1 Tool Crib 0.00 0.00 1998-06-01 00:00:00.000 2 2 Sheet Metal Racks 0.00 0.00 1998-06-01 00:00:00.000 3 3 Paint Shop 0.00 0.00 1998-06-01 00:00:00.000 4 4 Paint Storage 0.00 0.00 1998-06-01 00:00:00.000 5 5 Metal Storage 0.00 0.00 1998-06-01 00:00:00.000 6 6 Miscellaneous Storage 0.00 0.00 1998-06-01 00:00:00.000 7 7 Finished Goods Storage 0.00 0.00 1998-06-01 00:00:00.000 8 10 Frame Forming 22.50 96.00 1998-06-01 00:00:00.000 9 20 Frame Welding 25.00 108.00 1998-06-01 00:00:00.000 10 30 Debur and Polish 14.50 120.00 1998-06-01 00:00:00.000 11 40 Paint 15.75 120.00 1998-06-01 00:00:00.000 12 45 Specialized Paint 18.00 80.00 1998-06-01 00:00:00.000 13 50 Subassembly 12.25 120.00 1998-06-01 00:00:00.000 14 60 Final Assembly 12.25 120.00 1998-06-01 00:00:00.000 /* Note that we perform the same read as above when we were using the Repeatable Read transaction isolation level. */	
4		BEGIN TRANSACTION INSERT Production.Location (Name, Availability, CostRate, ModifiedDate) VALUES ('NewLoc1', 1.5, $5.00, GETDATE()) Results Messages /* Note that the Serializable Transaction Isolation Level does not allow the transaction in Session 2 to insert a new row into a range of data that has been read in the transaction in Session 1. REAL WORLD SCENARIO: (See step 2.) The second transaction cannot access inventory since all inventory records have been read by the first transaction. The results in step 1 are now accurate. */
5	ROLLBACK TRANSACTION	
		Messages (1 row(s) affected) /* Note that the Insert completes now that the transaction in Session 1 has completed. */

5. SNAPSHOT

Step	Session 1 (Query Window 1)	Session 2 (Query Window 2)

1	`USE AdventureWorks`
	`Go`
	
	`ALTER DATABASE AdventureWorks`
	`SET ALLOW_SNAPSHOT_ISOLATION ON`
	`GO`
	`/* Note that we must first turn on Snapshot Isolation on a database before it can be used in a transaction. This tells SQL Server to set up the virtual snapshot environment in TempDB. */`
	
	`SELECT EmployeeID, VacationHours`
	`FROM HumanResources.Employee`
	`WHERE EmployeeID = 4`
	Results | Messages
	EmployeeID VacationHours
	1 4 48
	
	`/* Note that the employee with EmployeeID 4 has 48 hours of vacation hours before any transactions begin. */`
	
	`SET TRANSACTION ISOLATION LEVEL SNAPSHOT`
	`Go`
	
	`BEGIN TRANSACTION`	

2		`USE AdventureWorks`
		`GO`
		
		`BEGIN TRANSACTION`
		
		`UPDATE HumanResources.Employee`
		`SET VacationHours = VacationHours - 8`
		`WHERE EmployeeID = 4`
		
		`SELECT VacationHours`
		`FROM HumanResources.Employee`
		`WHERE EmployeeID = 4`
		Results | Messages
		EmployeeID VacationHours
		1 4 40
		
		`/* Note that the update succeeded. Keep in mind that that change has not yet been committed. */`

3	`SELECT EmployeeID, VacationHours`
	`FROM HumanResources.Employee`
	`WHERE EmployeeID = 4`
	Results | Messages
	EmployeeID VacationHours
	1 4 48
	
	`/* Note that the update in Session 2 is not available here. This transaction continues to get its data from the snapshot taken when the transaction started. */`	

4		`COMMIT TRANSACTION` `GO`
5	`SELECT EmployeeID, VacationHours` `FROM HumanResources.Employee` `WHERE EmployeeID = 4` Results | Messages EmployeeID VacationHours 1 4 40 `/* Note that the changes committed in the transaction in Session 2 are still not available here. This transaction is still reading from the snapshot (versioned row in TempDB). In the next example, this will be different. */` `UPDATE HumanResources.Employee` `SET VacationHours = VacationHours + 8` `WHERE EmployeeID = 4` Messages Msg 3960, Level 16, State 2, Line 1 Snapshot isolation transaction aborted due to update conflict. `/* Note that this transaction is not allowed to even try to update data that have been updated in another transaction. When an attempt is made to do that, the transaction making the attempt is rolled back and terminated with an error message. */` `SELECT EmployeeID, VacationHours` `FROM HumanResources.Employee` `WHERE EmployeeID = 4` Results | Messages EmployeeID VacationHours 1 4 40 `/* Note that now that this transaction has terminated, we see the results of the update committed in session 1. */` `REAL WORLD SCENARIO: (See step 2 in Example 2.) After the second passenger has committed his reservation for seat 21C, the first passenger refreshes the screen to see if any changes have been made. The first passenger has no problem accessing the data, but the reservation made by the second passenger doesn't show up and seat 21C still shows available. When the first passenger tries to change her reserved seat to seat 21C, she is taken to the previous screen and a message informs her to try again at a later time.`  `*/`  `UPDATE HumanResources.Employee` `SET VacationHours = VacationHours + 8` `WHERE EmployeeID = 4` `/* Let's set the data back to its original state for the next example. */`	

B. Statement Isolation Level

6. READ COMMITTED SNAPSHOT

Step	Session 1 (Query Window 1)	Session 2 (Query Window 2)
1	USE AdventureWorks GO ALTER DATABASE AdventureWorks SET READ_COMMITTED_SNAPSHOT ON GO /* Note that we must first turn on Read Committed Snapshot on a database before it can be used. This tells SQL Server to set up the virtual snapshot environment in TempDB. */ SELECT EmployeeID, VacationHours FROM HumanResources.Employee WHERE EmployeeID = 4 Results | Messages EmployeeID VacationHours 1 4 48 /* Note that the employee with EmployeeID 4 has 48 hours of vacation hours before any transactions begin. */ SET TRANSACTION ISOLATION LEVEL READ COMMITTED GO BEGIN TRANSACTION	
2		USE AdventureWorks GO BEGIN TRANSACTION UPDATE HumanResources.Employee SET VacationHours = VacationHours - 8 WHERE EmployeeID = 4 SELECT VacationHours FROM HumanResources.Employee WHERE EmployeeID = 4 Results | Messages EmployeeID VacationHours 1 4 40 /* Note that the update succeeded. Keep in mind that that change has not yet been committed. */
3	SELECT EmployeeID, VacationHours FROM HumanResources.Employee WHERE EmployeeID = 4 Results | Messages EmployeeID VacationHours 1 4 48 /* Note that the update in Session 2 is not available here because the transaction in Session 2 has not yet committed. This transaction continues to get its data from the original snapshot. */	
4		COMMIT TRANSACTION GO
5	SELECT EmployeeID, VacationHours FROM HumanResources.Employee WHERE EmployeeID = 4 Results | Messages EmployeeID VacationHours 1 4 40 /* Note that the changes committed in Session 2 are now available here. The Read Committed Transaction Isolation Level allows this transaction to read committed data, whereas the Snapshot Isolation Level would still be reading from the versioned row in TempDB at this point. */ UPDATE HumanResources.Employee SET VacationHours = VacationHours + 8 WHERE EmployeeID = 4 Messages (1 row(s) affected) /* Note that this statement failed under Snapshot Isolation, because it attempts to modify data that have been modified by another transaction. But with Read Committed Snapshot Isolation, it succeeds. */ COMMIT TRANSACTION REAL WORLD SCENARIO: (See the previous example.) This time, when the first passenger refreshes the screen to see if any changes have been made, seat 21C is no longer available. */	

As you will see in the examples that follow, the higher the isolation level, the higher the level of protection (the more concurrency issues are prevented- Dirty read, Non repeatable read, Phantom read). Also, each isolation level includes the protections provided by the previous level so that each successively higher isolation level provides added protection in the form of more concurrency issues avoided. But, alas, nothing is free, and so the higher the isolation level, the less data availability there will be. Choosing the appropriate isolation level is a balancing act between highly safe concurrency and high data availability.