In this lab class we will approach the following topics:

1.1 - Transaction Execution in SQL Server

The main protocol adopted by SQL Server for managing concurrent transactions involves placing different types of locks on different database objects. SQL Server applies a write lock (i.e. exclusive lock) when it writes information, and a read lock (i.e. shared lock) when it reads information. Writing information usually refers to inserting, updating, or deleting rows, whereas reading information refers to retrieving rows with a SELECT statement.

The database objects that can be locked include **tables**, **database pages**, **rows**, and **index entries**. A database page is 8 KB in size, and any object resident within these 8 KB is locked implicitly when the database page is locked. Therefore, if a database page is locked, every row held on that page is effectively locked. Similarly, if a table is locked, every row in that table is locked.

In recent SQL Server versions, Microsoft has included the ability to use versioning, in addition to locking, to control the execution of concurrent transactions. In a system using versioning, data is not locked, but instead read operations are executed against an older version of the data being manipulated.

1.2 - Isolation Levels and Modifying the Default Locking Behavior

There are two ways in which SQL Server's default locking behavior can be modified. Individual SQL statements can be qualified with a keyword known as a **lock hint** to modify the locking behavior for that particular statement, or a default locking behavior for the connection can be set with the **SET TRANSACTION ISOLATION LEVEL** statement.

1.2.1 - Isolation levels

Levels of transaction isolation are specified by the ANSI SQL standard. Each level of isolation is associated to a different type of anomaly that may occur while concurrent transactions are running. The anomalies that may occur when allowing concurrent access to a database table are the following:

- **Dirty Reads** – A dirty read occurs when a transaction reads a data item that has been modified by another transaction not yet committed.

- **Non-repeatable Reads** – Non-repeatable reads happen when a query returns data that would be different if the query were repeated within the same transaction. Non-repeatable reads can occur when other transactions are modifying (committing) data that a transaction is reading (e.g. read locks are not acquired when performing a SELECT).

- **Phantoms** – The phantom problem occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first one. This can occur when a concurrent transaction is inserting tuples in the same table as the first one, and range locks are not correctly acquired on performing a SELECT.

The higher the isolation level, the more stringent the locking protocol – with the higher levels being a superset of the lower levels. The transaction isolation levels are as follows and, by default, SQL Server runs at **read committed** transaction isolation level.

Isolation Level	Dirty Reads	Non-repeatable Reads	Phantoms
Serializable	No	No	No
Repeatable Read	No	No	Yes
Read Committed	No	Yes	Yes
Read Uncommitted	Yes	Yes	Yes
Snapshot	No	No	No
Read Committed Snapshot	No	Yes	Yes

The first four isolation levels from the table above are implemented through a locking protocol, whereas the last two isolation levels are based on versioning.

1.2.2 - Lock hints

Lock hints are usually used on SELECT statements. The next line provides an example:

```
SELECT balance FROM accounts WITH (READUNCOMMITTED)
WHERE account_no = 1000;
```

The keywords available as lock hints, for modifying locking behavior, are as follows:

- **DBLOCK** - forces a shared database lock to be obtained when information is read.
- **HOLDLOCK** - forces a shared lock on a table to remain until the transaction completes. Key range locking will also be used to prevent phantom inserts. Nonrepeatable reads are also prevented.

- **NOLOCK** - allows a dirty read to take place— that is, a transaction can read the uncommitted changes made by another transaction.
- **PAGLOCK** - forces shared page locks to be taken where otherwise SQL Server may have used a table or row lock.
- **READCOMMITTED** - ensures that the statement behaves in the same way as if the connection were set to transaction isolation level READ COMMITTED.
- **READPAST** - enables a statement to skip rows that are locked by other statements.
- **READUNCOMMITTED** - equivalent to the NOLOCK lock hint.
- **REPEATABLEREAD** - ensures that the statement behaves in the same way as if the connection were set to transaction isolation level REPEATABLE READ.
- **ROWLOCK** - forces the use of rowlocks and is similar in use to PAGLOCK.
- **SERIALIZABLE** - forces shared locks to stay until the transaction completes. This is equivalent to specifying the HOLDLOCK hint.
- **TABLOCK** - forces a shared table lock to be taken where otherwise SQL Server may have used row locks.
- **UPDLOCK** - forces SQL Server to take update locks where otherwise SQL Server would have used shared locks.
- **XLOCK** - forces exclusive locks to be taken. This is typically used with TABLOCK and PAGLOCK.

Note: Beware of the **SET IMPLICIT_TRANSACTIONS ON** statement. It will automatically start a transaction when Transact-SQL statements such as SELECT, INSERT, UPDATE, and DELETE are used. The transaction will not be committed and its locks will not be released until an explicit COMMIT TRANSACTION statement is executed.[1]

1.3. Details About Locking and Versioning

We shall now analyze in more detail the intrinsics of locking and versioning in SQL Server.

Lock granularity

Applications require different levels of *lock granularity*. One application may benefit from page-level locking, while another application may benefit from row-level locking.

Locking individual columns provides the highest level of concurrency. By this, we mean that multiple users could be updating different columns in the same row simultaneously and they would not be involved in a lock conflict. If the lock granularity is implemented at the database level, the lowest level of concurrency is achieved. Multiple users could not simultaneously change anything at all in the database. However, the finer the granularity, the more system resources are used. This is why SQL Server and database systems in general do not lock at the column level. The amount of system resources used to manage all the locks would be too large.

[1] https://docs.microsoft.com/en-us/sql/t-sql/statements/set-implicit-transactions-transact-sql

Locks are approximately 100 bytes each in SQL Server. Locking at the column level would probably require tens of thousands of locks in a medium-sized database, which could equate to many megabytes of memory. The CPU resources needed to manage these locks would also be massive. Consequently, SQL Server locks rows, pages, and tables, which is a reasonable approach. The database itself can, of course, be set to single-user mode, which effectively provides locking at the database level.

Locking at the row level can be considered to be the default situation. One of the reasons that SQL Server tends to lock at the row level is that it has the capability to escalate locks, but not to de-escalate locks. Therefore, if SQL Server decides that a SQL statement is likely to lock the majority of rows in a table, it may lock at the table level. The same logic is used if SQL Server determines that most of the rows in a page are likely to be locked—it may take out a page lock instead of multiple row locks.

Intent Locks

SQL Server also makes use of a type of lock known as an *intent lock*. Intent locks are placed over tables and over pages in the tables, when a user locks rows in the table, and they stay in place for the life of the row locks. These locks are used primarily to ensure that a user cannot take locks on a table or pages in the table that would conflict with another user's row locks, without SQL Server having to check the actual existing row locks. For example, if a user was holding an exclusive row lock and another user wished to take an exclusive table lock on the table containing the row, the intent lock held on the table by the first user would ensure that its row lock would not be overlooked by the lock manager.

Deadlock prevention

Once a lock has been placed on an object, it has a lifetime. Suppose a T-SQL statement that causes a row lock is executed inside a user-defined transaction. It is possible in SQL Server to set a lock timeout value for a connection, so that transactions within this connection will **only wait** for their locks during a predefined period of time, after which the transaction will receive an error message informing it that the timeout period has been exceeded. In this context, a connection is a session of work, opened by a program (such as SQL Server Management Studio) to communicate with the database server, in order to execute transactions as a specific user.

A lock timeout value is set per connection as follows:

```
SET LOCK_TIMEOUT 10000
```

The timeout value is specified in milliseconds. The default value of -1 means wait indefinitely, whereas a value of 0 means do not wait at all.[2]

[2] https://docs.microsoft.com/en-us/sql/t-sql/statements/set-lock-timeout-transact-sql

Deadlock resolution

A *deadlock* situation can occur in SQL Server when a user holds a lock on a resource needed by another user who holds a lock on a resource needed by the first user. In this scenario, the users would wait forever if SQL Server did not intervene. SQL Server chooses one of the deadlocked users as a victim and issues a rollback for its transaction. A connection can set its deadlock priority such that, in the event of a transaction within this connection being involved in a deadlock, it will be chosen as the victim, as follows:

```
SET DEADLOCK_PRIORITY LOW
```

To return to the default deadlock handling mechanism, use the following code:

```
SET DEADLOCK_PRIORITY NORMAL
```

Generally, the transaction involved in the deadlock that has accumulated the least amount of CPU time is usually chosen as the victim.

Read, Write and Update locks

As well as placing shared and exclusive locks on database rows, SQL Server also makes use of a type of lock known as an *update lock*. These locks are associated with SQL statements that perform update and delete operations, which need to initially read rows before changing or deleting them. These rows have update locks placed on them that are compatible with shared read locks but are not compatible with other update locks or exclusive locks. If the rows must subsequently be updated or deleted, SQL Server attempts to promote the update locks to exclusive locks. If any other shared locks are associated with the rows, SQL Server will not be able to promote the update locks until these are released. Update locks are essentially an optimization to reduce deadlocks.

Snapshot isolation levels

Since SQL Server 2005, Microsoft has included the ability to use versioning in addition to locking (with `SET READ_COMMITTED_SNAPSHOT ON` and `SET TRANSACTION ISOLATION LEVEL SNAPSHOT`). In a system using versioning, data is not locked, but instead read operations happen on an older version of the data being manipulated. Depending upon isolation levels, the read operation can either read the latest committed data or the data as it was when the read operation started. Be aware that when an update operation takes place, the data being touched is copied to a separate storage area, incurring in a performance penalty (i.e., do not use versioning if you are doing large batch updates). Moreover, snapshot isolation will also permit **write skew** anomalies. In a write skew anomaly, two transactions (T1 and T2) concurrently read an overlapping data set (e.g. values V1 and V2), concurrently make disjoint updates (e.g. T1 updates V1, T2 updates V2), and finally concurrently commit, neither having seen the update performed by the other. If the isolation level was *serializable*, such an anomaly would be impossible, as either T1 or T2 would have to occur "first", and be visible to the other.

More details about the snapshot isolation level can be found online.[3]

2 Experiments and Exercises

2.1 - A Benchmark Experiment for SQL Server

In the following experiments, we will see how concurrent transactions are handled in SQL Server. For this purpose, we will have two users accessing the AdventureWorks database at the same time.

You can simulate these two users by opening two query windows in SQL Server Management Studio. Copy the commands of USER1 to one query window, and the commands of USER2 to the other query window. You can execute each command by selecting it and hitting Execute.

Case 1 : Using the read uncommitted isolation level - Users can query the value of a row/column that is in the middle of a transaction that has not yet committed. This corresponds to a dirty read anomaly.

USER 1	USER 2

`BEGIN TRANSACTION;`
 	
`UPDATE Production.Product`
`SET Name = 'Super Blade'`
`WHERE ProductID = 316;`
`-- was previously 'Blade', now 'Super Blade'`	

	`SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;`
	
	`BEGIN TRANSACTION;`
	
	`SELECT Name`
	`FROM Production.Product`
	`WHERE ProductID = 316;`
	`-- will return 'Super Blade', even before the first transaction commits`

`ROLLBACK TRANSACTION;`	

	`SELECT Name`
	`FROM Production.Product`
	`WHERE ProductID = 316;`
	`-- will now return 'Blade', the original value`
	
	`COMMIT TRANSACTION;`

Case 2 : Using the read committed isolation level – The transaction executed by USER 2 will wait on the SELECT statement until the COMMIT TRANSACTION of Transaction 1 is executed, because the READ COMMITTED isolation level can only read COMMITTED data. The dirty read anomaly therefore does not happen in this case.

[3] http://www.sqlteam.com/article/transaction-isolation-and-the-new-snapshot-isolation-level

USER 1	USER 2
BEGIN TRANSACTION; UPDATE Production.Product SET Name = 'Super Blade' WHERE ProductID = 316; -- was previously 'Blade', now 'Super Blade'	
	SET TRANSACTION ISOLATION LEVEL READ COMMITTED; BEGIN TRANSACTION; SELECT Name FROM Production.Product WHERE ProductID = 316; -- will wait until the first transaction commits
COMMIT TRANSACTION;	
	COMMIT TRANSACTION;

Execute the following statement to revert the product name to its original value:

```
UPDATE Production.Product SET Name = 'Blade' WHERE ProductID = 316;
```

In the READ COMMITTED isolation level, non-repeatable reads can still occur. In the next example, Transaction 2 will read different values for the same record:

USER 1	USER 2
	SET TRANSACTION ISOLATION LEVEL READ COMMITTED; BEGIN TRANSACTION; SELECT Name FROM Production.Product WHERE ProductID = 316; -- returns 'Blade'
BEGIN TRANSACTION; UPDATE Production.Product SET Name = 'Super Blade' WHERE ProductID = 316; -- was previously 'Blade', now 'Super Blade' COMMIT TRANSACTION;	
	SELECT Name FROM Production.Product WHERE ProductID = 316; -- returns 'Super Blade' COMMIT TRANSACTION;

Execute the following statement to revert the product name to its original value:

```
UPDATE Production.Product SET Name = 'Blade' WHERE ProductID = 316;
```

Case 3 : Using the repeatable read isolation level – In this case, Transaction 2 will always read the same piece of data and return the same result, not allowing Transaction 1's UPDATE to complete until Transaction 2 completes successfully.

USER 1	USER 2

	`SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;`
	
	`BEGIN TRANSACTION;`
	
	`SELECT Name`
	`FROM Production.Product`
	`WHERE ProductID = 316;`
	`-- returns 'Blade'`

`BEGIN TRANSACTION;`
 	
`UPDATE Production.Product`
`SET Name = 'Super Blade'`
`WHERE ProductID = 316;`
`-- will wait because user 2 has a lock on the row`	

	`SELECT Name`
	`FROM Production.Product`
	`WHERE ProductID = 316;`
	`-- still returns 'Blade'`
	
	`COMMIT TRANSACTION;`

`-- now it can proceed, since user 2 has committed` `ROLLBACK TRANSACTION;`	

When using the REPEATABLE READ isolation level, phantom reads can still occur. In the next example, we will read a different number of records within Transaction 2:

USER 1	USER 2

	`SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;`
	
	`BEGIN TRANSACTION;`
	
	`SELECT COUNT(*)`
	`FROM HumanResources.Department;`
	`-- returns number of rows in table`

`BEGIN TRANSACTION;`
 	
`INSERT INTO HumanResources.Department (Name,GroupName,ModifiedDate)`
`VALUES ('Test Department','G','9-04-2010');`
 	
`COMMIT TRANSACTION;`	

	`SELECT COUNT(*)`
	`FROM HumanResources.Department;`
	`-- returns a different number`
	
	`COMMIT TRANSACTION;`

Execute the following statement to delete the row that has been inserted by Transaction 1:

```
DELETE FROM HumanResources.Department WHERE Name = 'Test Department';
```

Case 4 : Using the serializable isolation level – This will lock out any insertions that would match the range, until the other transaction is committed, thus avoiding phantom reads.

USER 1	USER 2

	`SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`
	
	`BEGIN TRANSACTION;`
	
	`SELECT COUNT(*)`
	`FROM HumanResources.Department;`
	`-- returns number of rows in table`

`BEGIN TRANSACTION;`
 	
`INSERT INTO HumanResources.Department`
`(Name,GroupName,ModifiedDate)`
`VALUES ('Test Department','G','9-04-2010');`
`-- wait because this would affect rowcount for user 2`	

	`SELECT COUNT(*)`
	`FROM HumanResources.Department;`
	`-- still returns the same number`
	
	`COMMIT TRANSACTION;`

`-- now it can proceed, since user 2 has committed` `ROLLBACK TRANSACTION;`	

Case 5 : Using the snapshot isolation level – This is almost equivalent to using a SERIALIZABLE isolation level, although the execution is non-blocking. SQL Server does this by taking a "snapshot" (thus the name of the isolation level) into the **tempDB** database. This allows a developer to query against a specific version of the data throughout the entire transaction, without needing to wait for any other locks to clear and without reading any dirty data. The SNAPSHOT isolation level also avoids non-repeatable reads and phantom reads, but introduces an anomaly called “write skew”. In a write skew anomaly, two transactions (T1 and T2) concurrently read an overlapping data set (e.g. values V1 and V2), concurrently make disjoint updates (e.g. T1 updates V1, T2 updates V2), and finally concurrently commit, neither having seen the update performed by the other.

USER 1	USER 2
`ALTER DATABASE AdventureWorks2017` `SET ALLOW_SNAPSHOT_ISOLATION ON;`	
`SET TRANSACTION ISOLATION LEVEL SNAPSHOT;`	
`BEGIN TRANSACTION;`	
	`SET TRANSACTION ISOLATION LEVEL SNAPSHOT;`
	`BEGIN TRANSACTION;`

	`SELECT Name`
	`FROM Production.Product`
	`WHERE ProductID = 316;`
	`-- returns 'Blade'`

`UPDATE Production.Product`
`SET Name = 'Super Blade'`
`WHERE ProductID = 316;`
`-- was 'Blade', now 'Super Blade'`	

	`SELECT Name`
	`FROM Production.Product`
	`WHERE ProductID = 316;`
	`-- still 'Blade'`

`COMMIT TRANSACTION;`	

	`SELECT Name`
	`FROM Production.Product`
	`WHERE ProductID = 316;`
	`-- still 'Blade'`

	`COMMIT TRANSACTION;`

	`SELECT Name`
	`FROM Production.Product`
	`WHERE ProductID = 316;`
	`-- now it's 'Super Blade'`

Execute the following statement to revert the product name to its original value:

```
UPDATE Production.Product SET Name = 'Blade' WHERE ProductID = 316;
```

2.2 – Exercises

2.2.1. Consider the following 3 transactions and the schedule S involving the concurrent execution of the three transactions:

T1	**T2**	**T3**	**S**
begin	begin	begin	T3:read(z)
write(x)	read(x)	read(z)	T2:read(x)
read(y)	write(z)	write(y)	T1:write(x)
write(z)	commit	commit	T3:write(y)
commit			T1:read(y)
			T2:write(z)
			T1:write(z)

a) Present the precedence graph for S.

b) Is the schedule serializable? Justify and, if the schedule is serializable, indicate a corresponding serial schedule.

c) Consider the 2 Phase Locking (2PL) Protocol, in which one transaction obtains a lock on a data item before using it. Could schedule S be produced when using 2PL? Justify.

d) Consider the timestamp-based protocol for concurrency control, in which each transaction is issued a timestamp when it enters the system. Check if the schedule S would be allowed. Justify.

e) Consider again the timestamp-based protocol and the instructions on schedule S, but assume now that the timestamps associated to the transactions are T1 < T2 < T3. Check if the schedule S would be allowed. Justify.

2.2.2. Consider the following two transactions:

T1	**T2**
read(**A**)	read(**B**)
read(**B**)	read(**A**)
if **A** = 0 then **B** := **B** + 1	if **B** = 0 then **A** := **A** + 1
write(**B**)	write(**A**)

a) Add lock and unlock instructions to transactions **T**1 and **T**2, so that they observe the two-phase locking protocol.

b) Can the execution of these transactions result in a deadlock? Justify.