B.E (INFORMATION TECHNOLOGY)

ITPC406-DATABASE MANAGEMENT SYSTEM

STAFF NAME - Dr.K.RAJA/AP/IT/FEAT/AU

UNIT–IV

Transaction processing: Concurrency control, ACID property, Serializability of scheduling, Locking and timestamp based schedulers, Multi-version and optimistic Concurrency Control schemes, Database recovery.

UNIT–V

Database Security: Authentication, Authorization and access control, DAC, MAC and RBAC models, Intrusion detection, SQL injection.

Advanced topics: Object oriented and object relational databases, Logical databases, Web databases, Distributed databases, Data warehousing and data mining.

ACID properties in DBMS

To ensure the integrity of data during a transaction (**A transaction is a unit of program that updates various data items, read more about it here**), the database system maintains the following properties. These properties are widely known as ACID properties:

- **Atomicity**: This property ensures that either all the operations of a transaction reflect in database or none. Let's take an example of banking system to understand this: Suppose Account **A** has a balance of 400$ & **B** has 700$. Account **A** is transferring 100$ to Account **B**. This is a transaction that has two operations a) Debiting 100$ from A's balance b) Creating 100$ to B's balance. Let's say first operation passed successfully while second failed, in this case A's balance would be 300$ while B would be having 700$ instead of 800$. This is unacceptable in a banking system. Either the transaction should fail without executing any of the operation or it should process both the operations. The Atomicity property ensures that.
- **Consistency**: To preserve the consistency of database, the execution of transaction should take place in isolation (that means no other transaction should run concurrently when there is a transaction already running). For example account A is having a balance of 400$ and it is transferring 100$ to account B & C both. So we have two transactions here. Let's say these transactions run concurrently and both the transactions read 400$ balance,

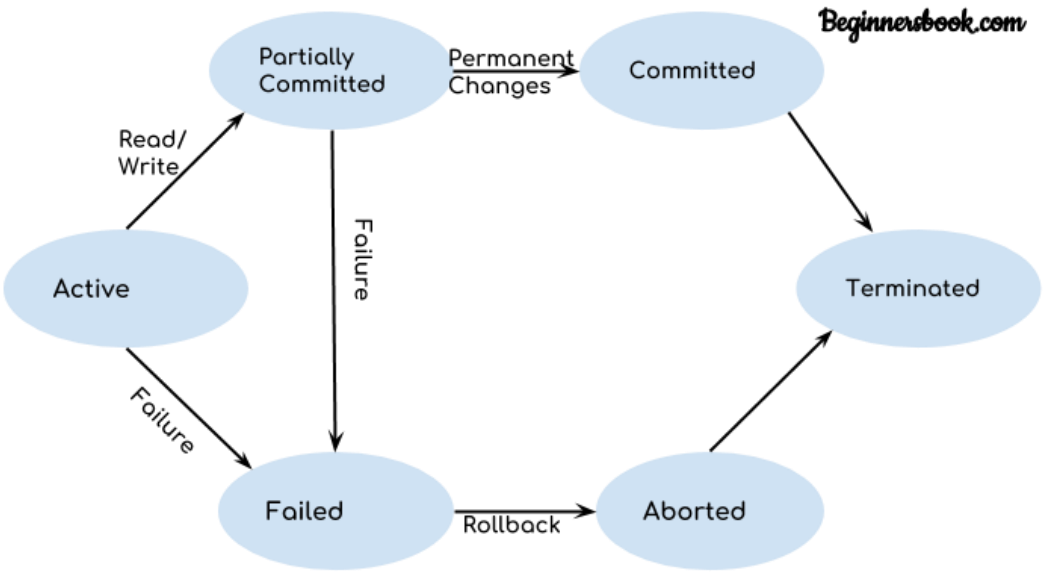
in that case the final balance of A would be 300$ instead of 200$. This is wrong. If the transaction were to run in isolation then the second transaction would have read the correct balance 300$ (before debiting 100$) once the first transaction went successful.

- **Isolation**: For every pair of transactions, one transaction should start execution only when the other finished execution. I have already discussed the example of Isolation in the Consistency property above.
- **Durability**: Once a transaction completes successfully, the changes it has made into the database should be permanent even if there is a system failure. The recovery-management component of database systems ensures the durability of transaction.

DBMS Transaction States

In this guide, we will discuss the **states of a transaction in DBMS**. A transaction in DBMS can be in one of the following states.

DBMS Transaction States Diagram



Lets discuss these states one by one.

Active State

As we have discussed in the DBMS transaction introduction that a transaction is a sequence of operations. If a transaction is in execution then it is said to be in active state. It doesn't matter which step is in execution, until unless the transaction is executing, it remains in active state.

Failed State

If a transaction is executing and a failure occurs, either a hardware failure or a software failure then the transaction goes into failed state from the active state.

Partially Committed State

As we can see in the above diagram that a transaction goes into “partially committed” state from the active state when there are read and write operations present in the transaction.

A transaction contains number of read and write operations. Once the whole transaction is successfully executed, the transaction goes into partially committed state where we have all the read and write operations performed on the main memory (local memory) instead of the actual database.

The reason why we have this state is because a transaction can fail during execution so if we are making the changes in the actual database instead of local memory, database may be left in an inconsistent state in case of any failure. **This state helps us to rollback the changes made to the database in case of a failure during execution.**

Committed State

If a transaction completes the execution successfully then all the changes made in the local memory during **partially committed** state are permanently stored in the database. You can also see in the above diagram that a transaction goes from partially committed state to committed state when everything is successful.

Aborted State

As we have seen above, if a transaction fails during execution then the transaction goes into a failed state. The changes made into the local memory (or buffer) are rolled back to the previous consistent state and the transaction goes into aborted state from the failed state. Refer the diagram to see the interaction between failed and aborted state.

DBMS Schedules and the Types of Schedules

We know that transactions are set of instructions and these instructions perform operations on database. When multiple transactions are running concurrently then there needs to be a sequence in which the operations are performed because at a time only one operation can be performed on the database. This sequence of operations is known as **Schedule**.

Lets take an example to understand what is a schedule in DBMS.

DBMS Schedule example

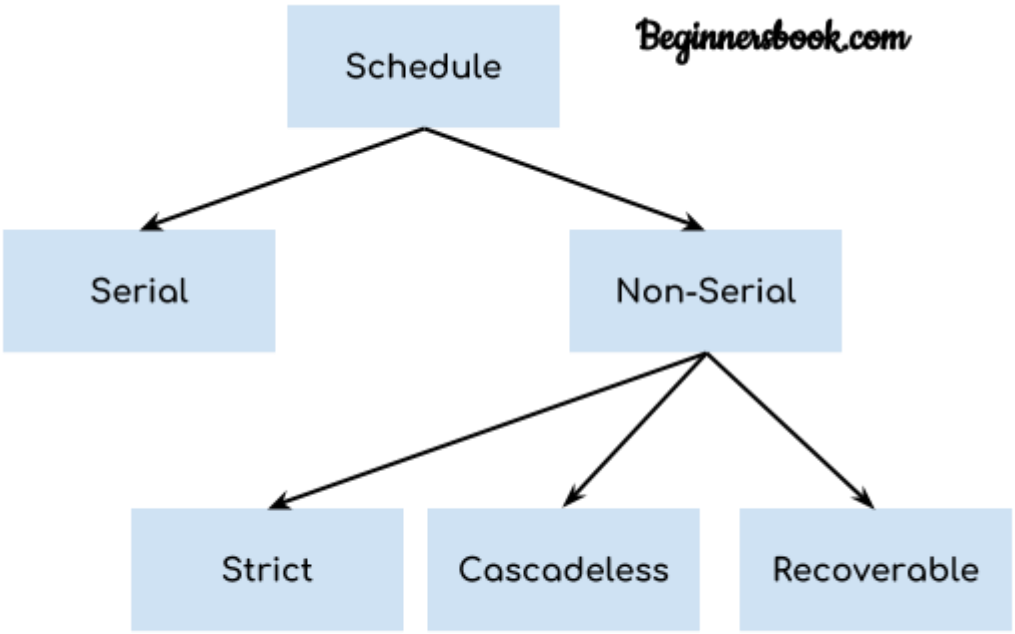
The following sequence of operations is a schedule. Here we have two transactions T1 & T2 which are running concurrently.

This schedule determines the exact order of operations that are going to be performed on database. In this example, all the instructions of transaction T1 are executed before the instructions of transaction T2, however this is not always necessary and we can have various types of schedules which we will discuss in this article.

```
T1        T2
----    ----
R(X)
W(X)
R(Y)
        R(Y)
        R(X)
        W(Y)
```

Types of Schedules in DBMS

We have various types of schedules in DBMS. Lets discuss them one by one.



Serial Schedule

In **Serial schedule**, a transaction is executed completely before starting the execution of another transaction. In other words, you can say that in serial schedule, a transaction does not start execution until the currently running transaction finished execution. This type of execution of transaction is also known as **non-interleaved** execution. The example we have seen above is the serial schedule.

Lets take another example.

Serial Schedule example
Here R refers to the read operation and W refers to the write operation. In this example, the transaction T2 does not start execution until the transaction T1 is finished.

```
T1        T2
----     ----
R(A)
R(B)
W(A)
commit
        R(B)
        R(A)
        W(B)
        commit
```

Strict Schedule

In Strict schedule, if the write operation of a transaction precedes a conflicting operation (Read or Write operation) of another transaction then the commit or abort operation of such transaction should also precede the conflicting operation of other transaction.

Lets take an example.

Strict Schedule example
Lets say we have two transactions Ta and Tb. The write operation of transaction Ta precedes the read or write operation of transaction Tb, so the commit or abort operation of transaction Ta should also precede the read or write of Tb.

```
Ta        Tb
-----    -----
R(X)
        R(X)
W(X)
commit
        W(X)
        R(X)
        commit
```

Here the write operation W(X) of Ta precedes the conflicting operation (Read or Write operation) of Tb so the conflicting operation of Tb had to wait the commit operation of Ta.

Cascadeless Schedule

In Cascadeless Schedule, if a transaction is going to perform read operation on a value, it has to wait until the transaction who is performing write on that value commits.

Cascadeless Schedule example

For example, lets say we have two transactions Ta and Tb. Tb is going to read the value X after the W(X) of Ta then Tb has to wait for the commit operation of transaction Ta before it reads the X.

```
Ta        Tb
-----    -----
R(X)
W(X)
        W(X)
commit
        R(X)
        W(X)
        commit
```

Recoverable Schedule

In Recoverable schedule, if a transaction is reading a value which has been updated by some other transaction then this transaction can commit only after the commit of other transaction which is updating value.

Recoverable Schedule example

Here Tb is performing read operation on X after the Ta has made changes in X using W(X) so Tb can only commit after the commit operation of Ta.

```
Ta        Tb
-----    -----
R(X)
W(X)
        R(X)
        W(X)
        R(X)
commit
        commit
```

DBMS Serializability

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.

What is a serializable schedule?

A serializable schedule always leaves the database in consistent state. A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability.

A non-serial schedule of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions. A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

Types of Serializability

There are two types of Serializability.

1. Conflict Serializability
2. View Serializability

DBMS Conflict Serializability

In the DBMS Schedules guide, we learned that there are two types of schedules – Serial & Non-Serial. A Serial schedule doesn't support concurrent execution of transactions while a non-serial schedule supports concurrency. We also learned in Serializability tutorial that a non-serial schedule may leave the database in inconsistent state so we need to check these non-serial schedules for the Serializability.

Conflict Serializability is one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not.

What is Conflict Serializability?

A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

Conflicting operations

Two operations are said to be in conflict, if they satisfy all the following three conditions:

1. Both the operations should belong to different transactions.
2. Both the operations are working on same data item.
3. At least one of the operation is a write operation.

Lets see some examples to understand this:

Example 1: Operation W(X) of transaction T1 and operation R(X) of transaction T2 are conflicting operations, because they satisfy all the three conditions mentioned above. They belong to different transactions, they are working on same data item X, one of the operation in write operation.

Example 2: Similarly Operations W(X) of T1 and W(X) of T2 are conflicting operations.

Example 3: Operations W(X) of T1 and W(Y) of T2 are non-conflicting operations because both the write operations are not working on same data item so these operations don't satisfy the second condition.

Example 4: Similarly R(X) of T1 and R(X) of T2 are non-conflicting operations because none of them is write operation.

Example 5: Similarly W(X) of T1 and R(X) of T1 are non-conflicting operations because both the operations belong to same transaction T1.

Conflict Equivalent Schedules

Two schedules are said to be conflict Equivalent if one schedule can be converted into other schedule after swapping non-conflicting operations.

Conflict Serializable check

Lets check whether a schedule is conflict serializable or not. If a schedule is conflict Equivalent to its serial schedule then it is called Conflict Serializable schedule. Lets take few examples of schedules.

Example of Conflict Serializability

Lets consider this schedule:

```
T1        T2
-----     ------
R(A)
R(B)
          R(A)
          R(B)
          W(B)
W(A)
```

To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1. However we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is **not Conflict Serializable**.

Lets take another example:

```
T1        T2
-----     ------
R(A)
          R(A)
          R(B)
          W(B)
R(B)
W(A)
```

Lets **swap non-conflicting operations**:

After swapping R(A) of T1 and R(A) of T2 we get:

```
T1        T2
-----     ------
          R(A)
R(A)
```

```
        R(B)
        W(B)
R(B)
W(A)
```
After swapping R(A) of T1 and R(B) of T2 we get:

```
T1        T2
-----     ------
        R(A)
        R(B)
R(A)
        W(B)
R(B)
W(A)
```
After swapping R(A) of T1 and W(B) of T2 we get:

```
T1        T2
-----     ------
        R(A)
        R(B)
        W(B)
R(A)
R(B)
W(A)
```
We finally got a serial schedule after swapping all the non-conflicting operations so we can say that the given schedule is **Conflict Serializable**.

DBMS View Serializability

In the last tutorial, we learned Conflict Serializability. In this article, we will discuss another type of serializability which is known as **View Serializability**.

What is View Serializability?

View Serializability is a process to find out that a given schedule is view serializable or not.

To check whether a given schedule is view serializable, we need to check whether the given schedule is **View Equivalent** to its serial schedule. Lets take an example to understand what I mean by that.

Given Schedule:

```
T1       T2
-----    ------
R(X)
W(X)
         R(X)
         W(X)
R(Y)
W(Y)
         R(Y)
         W(Y)
```

Serial Schedule of the above given schedule:
As we know that in Serial schedule a transaction only starts when the current running transaction is finished. So the serial schedule of the above given schedule would look like this:

```
T1       T2
-----    ------
R(X)
W(X)
R(Y)
W(Y)
         R(X)
         W(X)
         R(Y)
         W(Y)
```

If we can prove that the given schedule is **View Equivalent** to its serial schedule then the given schedule is called **view Serializable**.

Why we need View Serializability?

We know that a serial schedule never leaves the database in inconsistent state because there are no concurrent transactions execution. However a non-serial schedule can leave the database in inconsistent state because there are multiple transactions running concurrently. By checking that a given non-serial schedule is view serializable, we make sure that it is a consistent schedule.

You may be wondering instead of checking that a non-serial schedule is serializable or not, can't we have serial schedule all the time? The answer is no, because concurrent execution of transactions fully utilize the system resources and are considerably faster compared to serial schedules.

View Equivalent

Lets learn how to check whether the two schedules are view equivalent.

Two schedules T1 and T2 are said to be view equivalent, if they satisfy all the following conditions:

1. **Initial Read:** Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

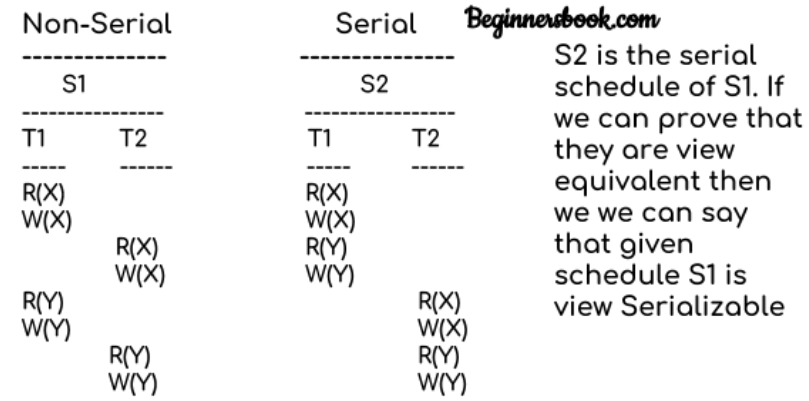
Read vs Initial Read: You may be confused by the term initial read. Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read. This will be more clear once we will get to the example in the next section of this same article.

2. **Final Write:** Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

3. **Update Read:** If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.View Serializable

If a schedule is view equivalent to its serial schedule then the given schedule is said to be View Serializable. Lets take an example.

View Serializable Example



Lets check the three conditions of view serializability:

Initial Read

In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.

Lets check for Y. In schedule S1, transaction T1 first reads the data item Y. In S2 also the first read operation on Y is performed by T1.

We checked for both data items X & Y and the **initial read** condition is satisfied in S1 & S2.

Final Write

In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.

Lets check for Y. In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.

We checked for both data items X & Y and the **final write** condition is satisfied in S1 & S2.

Update Read

In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.

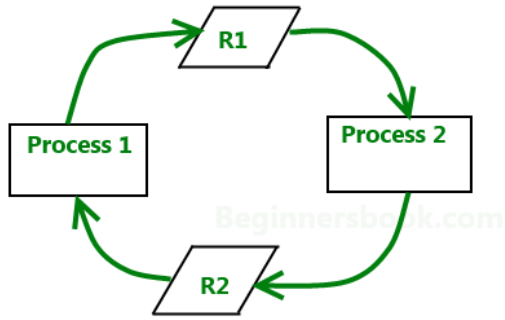
In S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.

The update read condition is also satisfied for both the schedules.

Result: Since all the three conditions that checks whether the two schedules are view equivalent are satisfied in this example, which means S1 and S2 are view equivalent. Also, as we know that the schedule S2 is the serial schedule of S1, thus we can say that the schedule S1 is view serializable schedule.

Deadlock in DBMS

A **deadlock** is a condition wherein two or more tasks are waiting for each other in order to be finished but none of the task is willing to give up the resources that other task needs. In this situation no task ever gets finished and is in waiting state forever.



Coffman conditions

Coffman stated four conditions for a deadlock occurrence. A deadlock may occur if all the following conditions holds true.

- **Mutual exclusion condition**: There must be at least one resource that cannot be used by more than one process at a time.
- **Hold and wait condition**: A process that is holding a resource can request for additional resources that are being held by other processes in the system.
- **No preemption condition**: A resource cannot be forcibly taken from a process. Only the process can release a resource that is being held by it.
- **Circular wait condition**: A condition where one process is waiting for a resource that is being held by second process and second process is waiting for third process ….so on and the last process is waiting for the first process. Thus making a circular chain of waiting.

Deadlock Handling

Ignore the deadlock (Ostrich algorithm)

Did that made you laugh? You may be wondering how ignoring a deadlock can come under deadlock handling. But to let you know that the windows you are using on your PC, uses this approach of deadlock handling and that is reason sometimes it hangs up and you have to reboot it to get it working. Not only Windows but UNIX also uses this approach.

The question is why? Why instead of dealing with a deadlock they ignore it and why this is being called as Ostrich algorithm?

Well! Let me answer the second question first, This is known as Ostrich algorithm because in this approach we ignore the deadlock and pretends that it would never occur, just like Ostrich behavior "to stick one's head in the sand and pretend there is no problem."

Let's discuss why we ignore it: When it is believed that deadlocks are very rare and cost of deadlock handling is higher, in that case ignoring is better solution than handling it. For example: Let's take the operating system example – If the time requires handling the deadlock is higher than the time requires rebooting the windows then rebooting would be a preferred choice considering that deadlocks are very rare in windows.

Deadlock detection

Resource scheduler is one that keeps the track of resources allocated to and requested by processes. Thus, if there is a deadlock it is known to the resource scheduler. This is how a deadlock is detected.

Once a deadlock is detected it is being corrected by following methods:

- **Terminating processes involved in deadlock**: Terminating all the processes involved in deadlock or terminating process one by one until deadlock is resolved can be the solutions but both of these approaches are not good. Terminating all processes cost high and partial work done by processes gets lost. Terminating one by one takes lot of time because each time a process is terminated, it needs to check whether the deadlock is resolved or not. Thus, the best approach is considering process age and priority while terminating them during a deadlock condition.
- **Resource Preemption**: Another approach can be the preemption of resources and allocation of them to the other processes until the deadlock is resolved.

Deadlock prevention

We have learnt that if all the four Coffman conditions hold true then a deadlock occurs so preventing one or more of them could prevent the deadlock.

- **Removing mutual exclusion**: All resources must be sharable that means at a time more than one processes can get a hold of the resources. That approach is practically impossible.

- **Removing hold and wait condition**: This can be removed if the process acquires all the resources that are needed before starting out. Another way to remove this to enforce a rule of requesting resource when there are none in held by the process.
- **Preemption of resources**: Preemption of resources from a process can result in rollback and thus this needs to be avoided in order to maintain the consistency and stability of the system.
- **Avoid circular wait condition**: This can be avoided if the resources are maintained in a hierarchy and process can hold the resources in increasing order of precedence. This avoid circular wait. Another way of doing this to force one resource per process rule – A process can request for a resource once it releases the resource currently being held by it. This avoids the circular wait.

Deadlock Avoidance

Deadlock can be avoided if resources are allocated in such a way that it avoids the deadlock occurrence. There are two algorithms for deadlock avoidance.

- Wait/Die
- Wound/Wait

Here is the table representation of resource allocation for each algorithm. Both of these algorithms take process age into consideration while determining the best possible way of resource allocation for deadlock avoidance.

	Wait/Die	Wound/Wait
Older process needs a resource held by younger process	**Older process** waits	**Younger process** dies
Younger process needs a resource held by older process	**Younger process** dies	**Younger process** waits

Once of the famous deadlock avoidance algorithm is **Banker's algorithm**

Concurrency Control in DBMS

When more than one transactions are running simultaneously there are chances of a conflict to occur which can leave database to an inconsistent state. To handle these conflicts we need concurrency control in DBMS, which allows transactions to run simultaneously but handles them in such a way so that the integrity of data remains intact.

Let's take an example to understand what I'm talking here.

Conflict Example

You and your brother have a joint bank account, from which you both can withdraw money. Now let's say you both go to different branches of the same bank at the same time and try to withdraw 5000 INR, your joint account has only 6000 balance. Now if we don't have concurrency control in place you both can get 5000 INR at the same time but once both the transactions finish the account balance would be -4000 which is not possible and leaves the database in inconsistent state.
We need something that controls the transactions in such a way that allows the transaction to run concurrently but maintaining the consistency of data to avoid such issues.

Solution of Conflicts: Locks

A lock is kind of a mechanism that ensures that the integrity of data is maintained. There are two types of a lock that can be placed while accessing the data so that the concurrent transaction can not alter the data while we are processing it.

1. Shared Lock(S)
2. Exclusive Lock(X)

1. Shared Lock(S): Shared lock is placed when we are reading the data, multiple shared locks can be placed on the data but when a shared lock is placed no exclusive lock can be placed.

For example, when two transactions are reading Steve's account balance, let them read by placing shared lock but at the same time if another transaction wants to update the Steve's account balance by placing Exclusive lock, do not allow it until reading is finished.

2. Exclusive Lock(X): Exclusive lock is placed when we want to read and write the data. This lock allows both the read and write operation, Once this lock is

placed on the data no other lock (shared or Exclusive) can be placed on the data until Exclusive lock is released.

For example, when a transaction wants to update the Steve's account balance, let it do by placing X lock on it but if a second transaction wants to read the data(S lock) don't allow it, if another transaction wants to write the data(X lock) don't allow that either.

So based on this we can create a table like this:

Lock Compatibility Matrix

```
_________________________
|    |  S   |   X   |
|-------------------------
| S  | True  |  False |
|-------------------------
| X  | False |  False |
--------------------------
```

How to read this matrix?:
There are two rows, first row says that when S lock is placed, another S lock can be acquired so it is marked true but no Exclusive locks can be acquired so marked False.
In second row, When X lock is acquired neither S nor X lock can be acquired so both marked false.

DBMS Concurrency Control: Two Phase, Timestamp, Lock-Based Protocol

What is Concurrency Control?

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another. Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical database, would have a mix of reading and WRITE operations and hence the concurrency is a challenge.

Concurrency control is used to address such conflicts which mostly occur with a multi-user system. It helps you to make sure that database transactions are performed concurrently without violating the data integrity of respective databases.

Therefore, concurrency control is a most important element for the proper functioning of a system where two or multiple database transactions that require access to the same data, are executed simultaneously.

In this tutorial, you will learn

Potential problems of Concurrency

Here, are some issues which you will likely to face while using the Concurrency Control method:

- **Lost Updates** occur when multiple transactions select the same row and update the row based on the value selected
- Uncommitted dependency issues occur when the second transaction selects a row which is updated by another transaction (**dirty read**)
- **Non-Repeatable Read** occurs when a second transaction is trying to access the same row several times and reads different data each time.
- **Incorrect Summary issue** occurs when one transaction takes summary over the value of all the instances of a repeated data-item, and second transaction update few instances of that specific data-item. In that situation, the resulting summary does not reflect a correct result.

Why use Concurrency method?

Reasons for using Concurrency control method is DBMS:

- To apply Isolation through mutual exclusion between conflicting transactions
- To resolve read-write and write-write conflict issues
- To preserve database consistency through constantly preserving execution obstructions
- The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
- Concurrency control helps to ensure serializability

Example

Assume that two people who go to electronic kiosks at the same time to buy a movie ticket for the same movie and the same show time.

However, there is only one seat left in for the movie show in that particular theatre. Without concurrency control, it is possible that both moviegoers will end up purchasing a ticket. However, concurrency control method does not allow this to happen. Both moviegoers can still access information written in the movie seating database. But concurrency control only provides a ticket to the buyer who has completed the transaction process first.

Concurrency Control Protocols

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose.

- Lock-Based Protocols
- Two Phase
- Timestamp-Based Protocols
- Validation-Based Protocols

Lock-based Protocols

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks: A Binary lock on a data item can either locked or unlocked states.

Shared/exclusive: This type of locking mechanism separates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

3. Simplistic Lock Protocol

This type of lock-based protocols allows transactions to obtain a lock on every object before beginning operation. Transactions may unlock the data item after finishing the 'write' operation.

4. Pre-claiming Locking

Pre-claiming lock protocol helps to evaluate operations and create a list of required data items which are needed to initiate an execution process. In the situation when all locks are granted, the transaction executes. After that, all locks release when all of its operations are over.

Starvation

Starvation is the situation when a transaction needs to wait for an indefinite period to acquire a lock.

Following are the reasons for Starvation:

- When waiting scheme for locked items is not properly managed
- In the case of resource leak
- The same transaction is selected as a victim repeatedly

Deadlock

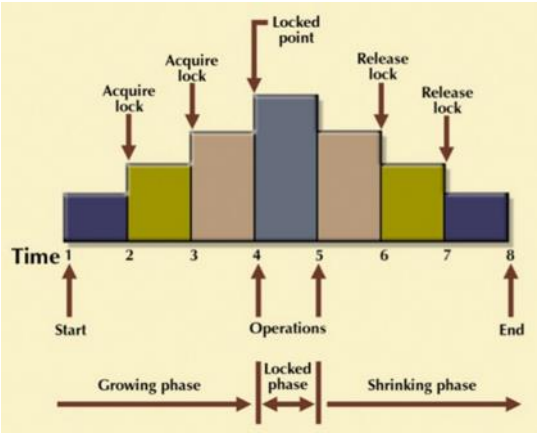
Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.

Two Phase Locking (2PL) Protocol

Two-Phase locking protocol which is also known as a 2PL protocol. It is also called P2L. In this type of locking protocol, the transaction should acquire a lock after it releases one of its locks.

This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.



The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

In the above-given diagram, you can see that local and global deadlock detectors are searching for deadlocks and solve them with resuming transactions to their initial states.

Strict Two-Phase Locking Method

Strict-Two phase locking system is almost similar to 2PL. The only difference is that Strict-2PL never releases a lock after using it. It holds all the locks until the commit point and releases all the locks at one go when the process is over.

Centralized 2PL

In Centralized 2 PL, a single site is responsible for lock management process. It has only one lock manager for the entire DBMS.

Primary copy 2PL

Primary copy 2PL mechanism, many lock managers are distributed to different sites. After that, a particular lock manager is responsible for managing the lock for a set of data items. When the primary copy has been updated, the change is propagated to the slaves.

Distributed 2PL

In this kind of two-phase locking mechanism, Lock managers are distributed to all sites. They are responsible for managing locks for data at that site. If no data is replicated, it is equivalent to primary copy 2PL. Communication costs of Distributed 2PL are quite higher than primary copy 2PL

Timestamp-based Protocols

The timestamp-based algorithm uses a timestamp to serialize the execution of concurrent transactions. This protocol ensures that every conflicting read and write operations are executed in timestamp order. The protocol uses the **System Time or Logical Count as** a Timestamp.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Example:

Suppose there are there transactions T1, T2, and T3.
T1 has entered the system at time 0010
T2 has entered the system at 0020
T3 has entered the system at 0030
Priority will be given to transaction T1, then transaction T2 and lastly Transacti on T3.

Advantages:

- Schedules are serializable just like 2PL protocols
- No waiting for the transaction, which eliminates the possibility of deadlocks!

Disadvantages:

Starvation is possible if the same transaction is restarted and continually aborted

Characteristics of Good Concurrency Protocol

An ideal concurrency control DBMS mechanism has the following objectives:

- Must be resilient to site and communication failures.
- It allows the parallel execution of transactions to achieve maximum concurrency.
- Its storage mechanisms and computational methods should be modest to minimize overhead.
- It must enforce some constraints on the structure of atomic actions of transactions.

Summary

- Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another.
- Lost Updates, dirty read, Non-Repeatable Read, and Incorrect Summary Issue are problems faced due to lack of concurrency control.
- Lock-Based, Two-Phase, Timestamp-Based, Validation-Based are types of Concurrency handling protocols
- The lock could be Shared (S) or Exclusive (X)
- Two-Phase locking protocol which is also known as a 2PL protocol needs transaction should acquire a lock after it releases one of its locks. It has 2 phases growing and shrinking.

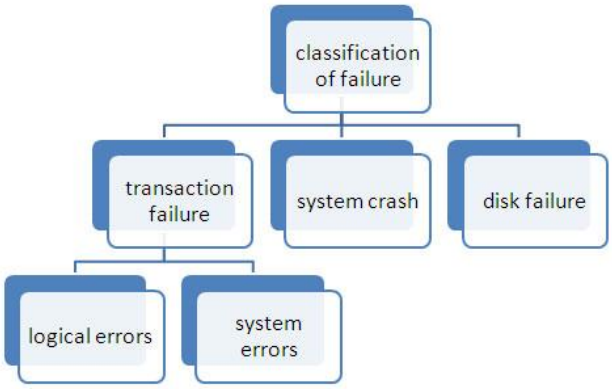
- The timestamp-based algorithm uses a timestamp to serialize the execution of concurrent transactions. The protocol uses the **System Time or Logical Count as** a Timestamp.

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

Multiversion 2 phase locking:
Each successful write results in the creation of a new version of the data item written. Timestamps are used to label the versions. When a read(X) operation is issued, select an appropriate version of X based on the timestamp of the transaction.
The optimistic concurrency control (MVOCC) algorithm is based on the assumption that transactions are unlikely to conflict. They are split into three phases: read, validation, and write, and the protocol minimizes the time that a transaction holds locks on tuples. This protocol is used in MemSQL, HyPer, and MS Hekaton.

DATABASE RECOVERY IN DBMS AND ITS TECHNIQUES: There can be any case in database system like any computer system when database failure happens. So data stored in database should be available all the time whenever it is needed. So Database recovery means recovering the data when it get deleted, hacked or damaged accidentally. Atomicity is must whether is transaction is over or not it should reflect in the database permanently or it should not effect the database at all. So database recovery and database recovery techniques are must in DBMS. So database recovery techniques in DBMS are given below.
Also See: Keys in DBMS
Crash recovery:
DBMS may be an extremely complicated system with many transactions being executed each second. The sturdiness and hardiness of software rely upon its complicated design and its underlying hardware and system package. If it fails or crashes amid transactions, it's expected that the system would follow some style of rule or techniques to recover lost knowledge.
DATABASE RECOVERY IN DBMS AND ITS TECHNIQUES
Classification of failure:
To see wherever the matter has occurred, we tend to generalize a failure into numerous classes, as follows:

- Transaction failure
- System crash
- Disk failure



Types of Failure

1. **Transaction failure:** A transaction needs to abort once it fails to execute or once it reaches to any further extent from wherever it can't go to any extent further. This is often known as transaction failure wherever solely many transactions or processes are hurt. The reasons for transaction failure are:

- Logical errors
- System errors

1. **Logical errors:** Where a transaction cannot complete as a result of its code error or an internal error condition.
2. **System errors:** Wherever the information system itself terminates an energetic transaction as a result of the DBMS isn't able to execute it, or it's to prevent due to some system condition. to Illustrate, just in case of situation or resource inconvenience, the system aborts an active transaction.
3. **System crash:** There are issues − external to the system − that will cause the system to prevent abruptly and cause the system to crash. For instance, interruptions in power supply might cause the failure of underlying hardware or software package failure. Examples might include OS errors.
4. **Disk failure:** In early days of technology evolution, it had been a typical drawback wherever hard-disk drives or storage drives accustomed to failing oftentimes. Disk failures include the formation of dangerous sectors, unreachability to the disk, disk crash or the other failure, that destroys all or a section of disk storage.

Database SecurityDatabase security has many different layers, but the key aspects are:

Authentication

User authentication is to make sure that the person accessing the database is who he claims to be. Authentication can be done at the operating system level or even the database level itself. Many authentication systems such as retina scanners or bio-metrics are used to make sure unauthorized people cannot access the database.

Authorization

Authorization is a privilege provided by the Database Administer. Users of the database can only view the contents they are authorized to view. The rest of the database is out of bounds to them.

The different permissions for authorizations available are:

- **Primary Permission -** This is granted to users publicly and directly.
- **Secondary Permission -** This is granted to groups and automatically awarded to a user if he is a member of the group.
- **Public Permission -** This is publicly granted to all the users.
- **Context sensitive permission -** This is related to sensitive content and only granted to a select users.

The categories of authorization that can be given to users are:

- **System Administrator -** This is the highest administrative authorization for a user. Users with this authorization can also execute some database administrator commands such as restore or upgrade a database.
- **System Control -** This is the highest control authorization for a user. This allows maintenance operations on the database but not direct access to data.
- **System Maintenance -** This is the lower level of system control authority. It also allows users to maintain the database but within a database manager instance.
- **System Monitor -** Using this authority, the user can monitor the database and take snapshots of it.

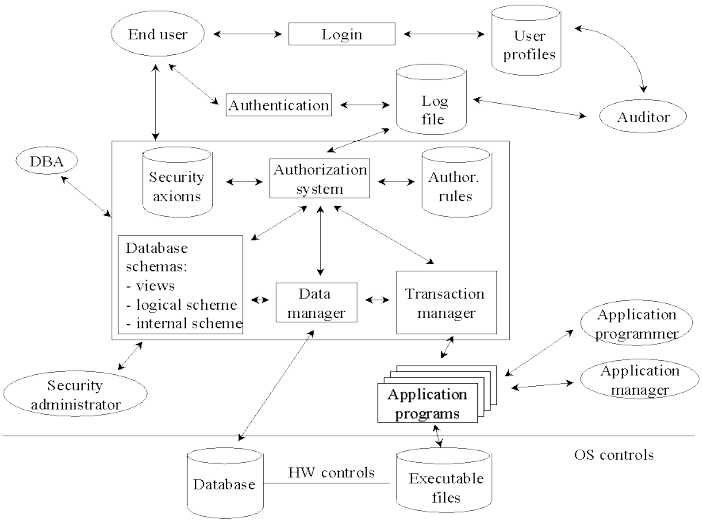
Database Integrity

Data integrity in the database is the correctness, consistency and completeness of data. Data integrity is enforced using the following three integrity constraints:

- **Entity Integrity -** This is related to the concept of primary keys. All tables should have their own primary keys which should uniquely identify a row and not be NULL.
- **Referential Integrity -** This is related to the concept of foreign keys. A foreign key is a key of a relation that is referred in another relation.
- **Domain Integrity -** This means that there should be a defined domain for all the columns in a database.

Access Control

Access control is responsible for control of rules determined by security policies for all direct accesses to the system. Traditional control systems work with notions *subject*, *object* and *operation*.



Authorization Models: ACL, DAC, MAC, RBAC, ABAC

ACL (Access Control List)

- Subject can Action to Object
- Base on user and group

Example

1. Granting Dino article created permission.

2. Subject: Dino
3. Action: Create
4. Object: Article

5. Dino can create article now.

DAC (Discretionary Access Control)

The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject.

- Subject can Action to Object
- Subject can grant other Subject

- Base on user and group

Example

1. Granting Dino article created permission.

2. Subject: Dino
3. Action: Create
4. Object: Article

5. Dino can create article now, and give this permission to others.
6. Dino grants James to create articles.

7. Subject: James
8. Action: Create
9. Object: Article

10. James can create article now.

MAC (Mandatory Access Control)

Subjects and objects each have a set of security attributes. Whenever a subject attempts to access an object, an authorization rule enforced by the operating system kernel examines these security attributes and decides whether the access can take place. Any operation by any subject on any object is tested against the set of authorization rules (aka policy) to determine if the operation is allowed.

With mandatory access control, this security policy is centrally controlled by a security policy administrator; users do not have the ability to override the policy and, for example, grant access to files that would otherwise be restricted.

- Subject can Action to Object
- Object can be Action by Subject
- Base on user and group

Example

1. Granting Dino article created permission.

2. Subject: Dino
3. Action: Create

4. Object: Article

5. Let Article could be created by Dino.

6. Subject: Article
7. Action: Created
8. Object: Dino

9. Dino can create article now.

RBAC (Role-Based Access Control)

RBAC differs from access control lists (ACLs), used in traditional discretionary access-control systems, in that **it assigns permissions to specific operations with meaning in the organization**, rather than to low level data objects. For example, an access control list could be used to grant or deny write access to a particular system file, but it would not dictate how that file could be changed. In an RBAC-based system, an operation might be to 'create a credit account' transaction in a financial application or to 'populate a blood sugar level test' record in a medical application.

- Subject is a Role which has Permission of Action to Object
- Can implement mandatory access control (MAC) or discretionary access control (DAC).
- (User or group)-role-permission-object
- Concept
    - Subject
    - Role
    - Permission
    - Operation

Group vs Role

- Group: a collection of users
    - Dino, James and Liam are members of Meifamly Organization.
- Role: a collection of permissions
    - Writer is a role, which can create, update articles.
    - Role can be applied to user and group.

Example

1. Set permissions named write article and manage article

```
Permission:
  - Name: write article
  - Operations:
    - Object: Article
      Action: Created
    - Object: Article
      Action: Updated
    - Object: Article
      Action: Read
  - Name: manage article
  - Operations:
    - Object: Article
      Action: Delete
    - Object: Article
      Action: Read
```

17. Set a Role named Writer, give it write article permission, and a Role named Manager, give it manage article permission. CEO has all permissions.

```
Role:
  - Name: Writer
    Permissions:
      - write article
  - Name: Manager
    Permissions:
      - manage article
  - Name: CEO
    Permissions:
      - write article
      - manage article
```

29. Give Dino Writer role

```
Subject: Dino
Roles:
  - Writer
```

33. Dino can create article now.

34. Give James Writer and Manager roles

```
Subject: James
Roles:
  - Writer
  - Manager
```

James can create and delete article now.

Intrusion Detection System (IDS)

An **Intrusion Detection System (IDS)** is a system that monitors **network traffic** for suspicious activity and issues alerts when such activity is discovered. It is a software application that scans a network or a system for harmful activity or policy breaching. Any malicious venture or violation is normally reported either to an administrator or collected centrally using a security information and event management (SIEM) system. A SIEM system integrates outputs from multiple sources and uses alarm filtering techniques to differentiate malicious activity from false alarms.
Although intrusion detection systems monitor networks for potentially malicious activity, they are also disposed to false alarms. Hence, organizations need to fine-tune their IDS products when they first install them. It means properly setting up the intrusion detection systems to recognize what normal traffic on the network looks like as compared to malicious activity.

Intrusion prevention systems also monitor network packets inbound the system to check the malicious activities involved in it and at once sends the warning notifications.

Classification of Intrusion Detection System:
IDS are classified into 5 types:

1. **Network Intrusion Detection System (NIDS):**
   Network intrusion detection systems (NIDS) are set up at a planned point within the network to examine traffic from all devices on the network. It performs an observation of passing traffic on the entire subnet and matches the traffic that is passed on the subnets to the collection of known attacks. Once an attack is identified or abnormal behavior is observed, the alert can be sent to the administrator. An example of an NIDS is installing it on the subnet where firewalls are located in order to see if someone is trying crack the firewall.
2. **Host Intrusion Detection System (HIDS):**
   Host intrusion detection systems (HIDS) run on independent hosts or devices on the network. A HIDS monitors the incoming and outgoing packets from the device only and will alert the administrator if suspicious or malicious activity is detected. It takes a snapshot of existing system files and compares it with the previous snapshot. If the analytical system files were

edited or deleted, an alert is sent to the administrator to investigate. An example of HIDS usage can be seen on mission critical machines, which are not expected to change their layout.

3. **Protocol-based Intrusion Detection System (PIDS):** Protocol-based intrusion detection system (PIDS) comprises of a system or agent that would consistently resides at the front end of a server, controlling and interpreting the protocol between a user/device and the server. It is trying to secure the web server by regularly monitoring the HTTPS protocol stream and accept the related HTTP protocol. As HTTPS is un-encrypted and before instantly entering its web presentation layer then this system would need to reside in this interface, between to use the HTTPS.
4. **Application Protocol-based Intrusion Detection System (APIDS):** Application Protocol-based Intrusion Detection System (APIDS) is a system or agent that generally resides within a group of servers. It identifies the intrusions by monitoring and interpreting the communication on application specific protocols. For example, this would monitor the SQL protocol explicit to the middleware as it transacts with the database in the web server.
5. **Hybrid Intrusion Detection System :** Hybrid intrusion detection system is made by the combination of two or more approaches of the intrusion detection system. In the hybrid intrusion detection system, host agent or system data is combined with network information to develop a complete view of the network system. Hybrid intrusion detection system is more effective in comparison to the other intrusion detection system. Prelude is an example of Hybrid IDS.

Detection Method of IDS:

1. **Signature-based Method:** Signature-based IDS detects the attacks on the basis of the specific patterns such as number of bytes or number of 1’s or number of 0’s in the network traffic. It also detects on the basis of the already known malicious instruction sequence that is used by the malware. The detected patterns in the IDS are known as signatures.
   Signature-based IDS can easily detect the attacks whose pattern (signature) already exists in system but it is quite difficult to detect the new malware attacks as their pattern (signature) is not known.

2. **Anomaly-based Method:**
   Anomaly-based IDS was introduced to detect the unknown malware attacks as new malware are developed rapidly. In anomaly-based IDS there is use of machine learning to create a trustful activity model and anything coming is compared with that model and it is declared suspicious if it is not found in model. Machine learning based method has a better generalized property in comparison to signature-based IDS as these models can be trained according to the applications and hardware configurations.

Comparison of IDS with Firewalls:
IDS and firewall both are related to the network security but an IDS differs from a firewall as a firewall looks outwardly for intrusions in order to stop them from happening. Firewalls restrict access between networks to prevent intrusion and if an attack is from inside the network it don't signal. An IDS describes a suspected intrusion once it has happened and then signals an alarm.

SQL Injection

SQL injection is a code injection technique that might destroy your database.

SQL injection is one of the most common web hacking techniques.

SQL injection is the placement of malicious code in SQL statements, via web page input.

SQL in Web Pages

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will **unknowingly** run on your database.

Look at the following example which creates a SELECT statement by adding a variable (txtUserId) to a select string. The variable is fetched from user input (getRequestString):

Example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

The rest of this chapter describes the potential dangers of using user input in SQL statements.

SQL Injection Based on 1=1 is Always True

Look at the example above again. The original purpose of the code was to create an SQL statement to select a user, with a given user id.

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

UserId: 105 OR 1=1

Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always TRUE.

Does the example above look dangerous? What if the "Users" table contains names and passwords?

The SQL statement above is much the same as this:

```
SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;
```

A hacker might get access to all the user names and passwords in a database, by simply inserting 105 OR 1=1 into the input field.

SQL Injection Based on ""="" is Always True

Here is an example of a user login on a web site:

Username:
John Doe

Password:
myPass

Example

```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

sql = 'SELECT * FROM Users WHERE Name ='" + uName + "' AND Pass ='" + uPass + ""
```

Result

```
SELECT * FROM Users WHERE Name ="John Doe" AND Pass ="myPass"
```

A hacker might get access to user names and passwords in a database by simply inserting " OR ""=" into the user name or password text box:

User Name:

Password:

The code at the server will create a valid SQL statement like this:

Result

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

The SQL above is valid and will return all rows from the "Users" table, since **OR** ""="" is always TRUE.

SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

Example

```
SELECT * FROM Users; DROP TABLE Suppliers
```

Look at the following example:

Example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

And the following input:

User id: 105; DROP T

The valid SQL statement would look like this:

Result

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

Use SQL Parameters for Protection

To protect a web site from SQL injection, you can use SQL parameters.

SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

ASP.NET Razor Example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = @0";
db.Execute(txtSQL,txtUserId);
```

Note that parameters are represented in the SQL statement by a @ marker.

The SQL engine checks each parameter to ensure that it is correct for its column and are treated literally, and not as part of the SQL to be executed.

Another Example

```
txtNam = getRequestString("CustomerName");
txtAdd = getRequestString("Address");
txtCit = getRequestString("City");
txtSQL = "INSERT INTO Customers (CustomerName,Address,City) Values(@0,@1,@2)";
db.Execute(txtSQL,txtNam,txtAdd,txtCit);
```

Examples

The following examples shows how to build parameterized queries in some common web languages.

SELECT STATEMENT IN ASP.NET:

```
txtUserId = getRequestString("UserId");
sql = "SELECT * FROM Customers WHERE CustomerId = @0";
command = new SqlCommand(sql);
command.Parameters.AddWithValue("@0",txtUserID);
command.ExecuteReader();
```

INSERT INTO STATEMENT IN ASP.NET:

```
txtNam = getRequestString("CustomerName");
txtAdd = getRequestString("Address");
txtCit = getRequestString("City");
txtSQL = "INSERT INTO Customers (CustomerName,Address,City) Values(@0,@1,@2)";
command = new SqlCommand(txtSQL);
```

```
command.Parameters.AddWithValue("@0",txtNam);
command.Parameters.AddWithValue("@1",txtAdd);
command.Parameters.AddWithValue("@2",txtCit);
command.ExecuteNonQuery();
```

INSERT INTO STATEMENT IN PHP:

```
$stmt = $dbh->prepare("INSERT INTO Customers
(CustomerName,Address,City)
VALUES (:nam, :add, :cit)");
$stmt->bindParam(':nam', $txtNam);
$stmt->bindParam(':add', $txtAdd);
$stmt->bindParam(':cit', $txtCit);
$stmt->execute();
```