A database system may be classified according to the number of users who can use the system concurrently. A DBMS is single-user if at most one user at a time can use the system and it is multiuser if many users can use the system concurrently. Single-user DBMSs are generally used in small applications that reside on personal computers (i.e., TinyDB in mobile apps). Multiuser databases accomplish this functionality by using interleaved concurrency. Database operations do not actually happen at the exact same time. They are instead interleaved which means while one operation is working the other is waiting its turn to work on the database.

A transaction is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations which can include insertion, deletion, modification or retrieval operations.

One way of specifying the transaction boundaries is by specifying explicit begin transaction and end transaction statements in an application program. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called read-only, otherwise it is known as a read-write transaction.

Several problems can occur when concurrent transactions execute in an uncontrolled manner. The Lost Update Problem occurs when two transactions that access the same database items their operations have their operations interleaved in a way that makes the value of some database items incorrect. On example of this problem would occur with bank accounts. If one transactions T1 reads the current amount, takes that amount and adds $100 to it. Before writing that value another transaction T2 reads the current amount, takes that amount and subtracts $50 from it. At this point, T1 writes the account balance + $100. After this T2 writes the account balance(original balance it had read before T1 had written its result) - $50. The resulting balance in the account will be lower than it is supposed to be by $100. The $100 deposit is lost.

The Temporary Update (or Dirty Read) problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the update item is accessed(read) by another transaction before it is changed back to its original value. In this problem imagine the $100 dollar deposit(T1) fails at some point but while it still processing another transaction(T2) reads the account balance. T2 will have a incorrect balance of the account since T1 fails and will be rolled back to its orginal value. T2 should not have been allowed to read the account balance until T1 completed or was rolled back.

The Incorrect Summary problem occurs if one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated. If I am retreiving a summary of my bank accounts and transactions are still completing in the midst of the construction of the summary, the summary will not be accurate.

The Unrepeatable Read problem occurs when a transaction T reads the same item twice and the item is changed by another transaction T’ between the two reads. Hence T receives different values for its two reads of the same item. If a transaction reads the balance to check if it is possible to make a withdrawl, then once it is confirmed that a withdrawl is possible it reads the balance again to actual perform the withdrawl but there is different value there than the one orginaly read.

Transactions should posses several properties often called ACID properties: Atomicity, Consitency preservation, Isolation, Durability or permanency. A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all. This will be enforced by transaction recovery subsystem. If a transaction fails to complete then the subsystem must undo any effects of the transaction on the database. On the other hand, write operations of a comitted transaction must be written to disk.

A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another. This usualy the responsibility of the programmers who write the database programs or of the DBMS moduel enforces integrity constraints.

A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently. This is enforced by the concurrency control subsystem.

The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure. This responsibility lies with the recovery subsystem of the DBMS.

The order in which varios transactions execute is known as the schedule or history. Schedules can be characterized in terms of their recoverability. A shorthand notation for describing a schedule uses the symbols b, r, w, e, c, and a for the operations begin\_transaction, read\_item, write\_item, end\_transaction, commit, and abort, respectively, and appends as a subscript the transaction id (transaction number) to each operation in the schedule.

Two operations in a schedule are in conflict if the belong to different transactions, they access the same item X, and at least one of the operations is a write\_item(X).

Schedules can be characterized as recoverable, cascadeless and strict. There is much overlap between all of these destinctions. Strict schedules are also cascadeless and cascadeles schedules are recoverable.

A schedule is serial if for every transactionT participating in the schedule, all the operations of T are exectuded consecutively in the schedule; otherwise, the schedule is called nonserial. A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions. A serial schedule is considered correct because every transaction is assumed to be correct if executed on its own. A serializable schedule is considered correct because it is equivalent to a serial schedule, which is considered correct.

Being serializable is distinct from being serial, however. A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for another transaction to terminate, thus slowing down processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness.

A binary lock enforces mutual exclusion on the data item. This system ends up being too restrictive, if every transaction is seeking to read the locked item, then there is not very good reason to lock the item since read operations from different transactions cannot conflict. Shared/Exclusive locks are favored as they allow any number of transactions to read the share-locked item while an exclusive-locked item is only accessible by one transaction that holds its lock.

The two-phase locking (2PL) protocol is followed if all locking operations (read\_lock, write\_lock) precede the first unlock operation in the transaction. If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable.

Conservative 2PL requires a transaction to lock all the items it accesses before the transaction begins execution. If any of the items needed cannot be locked, the transaction does not lock any item and instead waits until all items are available for locking. In strict 2PL a transaction does not release any of its exclusive locks until after it commits or aborts. This leads to a strict schedule for recoverability although it is not deadlock-free.

Deadlock occurs when each transaction in a set of two or more transactions is waiting for some item that is locked by some other transaction in the set. One way to prevent deadlock is to use a deadlock prevention protocol such as wait-die or wound-wait. Although both techniques are deadlock-free both may cause some transactions to abort needlessly.

Starvation occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. A fair waiting scheme such as FIFO can be used to mitigate starvation. Transactions are enabled to lock an item in the order in which they originally requested the lock.

Wait-die priotizes younger transactions and allows the older transaction to wait until the resource is available for execution. Wound-wait priotizies older transactions and allows the younger transaction to wait until the resource is available for execution.

Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting(NW) and cautious waiting(CW) algorithms.

In the no waiting algorithm, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whehther a deadlock will actually occur or not. However, this scheme can cause transactions to abort and restart needlessly.

The cautious waiting algorithm was proposed to try and reduce the number of needless aborts/restarts. Suppose that a transaction Ti tries to lock an item X but is not able to do so because it is locked by some other transaction Tj. If the Tj is not blocked then Ti is blocked and allowed to wait, otherwise abort Ti.

The use of timeouts is practical because of its low overhead and simplicity. If a transaction waits for a period longer thatn a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it.

Timestamp is a unique identifier created by the DBMS to identify a transaction by the order in which they were submitted to the system. One method for generating timestamps is to use a counter that is incremented each time its value is assigned to a transaction. Depending on the maximum size of the counter the system will need to reset this counter to zero when no transactions are executing. Another way to implement timestamps is to use the current date/time value from the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

In Timestamp Ordering protocol transactions are orderered using timestamps. Each item contains a read\_TS which is the youngest transaction to have read this item and write\_TS is the youngest transaction to have written to this item. With this information we can determine if transaction should be allowed to proceed. A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamps. The algorithm must insure that, for each item accessed by conflicting operations, the order in which the item is accessed does not violate the timestamp order. First we associate two timestamps, read\_TS is the youngest time stamp to have read this item and write\_TS is the youngest time stamp to have written to this item.

Basic Timestamp Ordering(basic TO) checks that when a transaction tires to issue a read\_item or write\_item operation the timestamps of read\_TS or write\_TS do not violate the order of transaction execution. If this order is violated then T is aborted and resubmitted to the system as new transaction with a new timestamp. If the transaction is aborted and rolled back, a cascading rollback may occur and is one of the problems with basic TO.

In basic TO, if the last succesful read of the item is older than the timestamp of the transaction or if the timestamp of the transaction is older than the last succesful write of the item then the transaction is aborted, rolled back and rejected. This means that a younger transaction has already read or written the value of the item and this current transaction will violate timestamp ordering.