ST. XAVIER’S COLLEGE

**Maitighar,Kathmandu**

**(Affiliated to Tribhuvan University)**



**Database Management System**

**Lab Assignment #11**

**Submitted By**

Aashish Raj Shrestha

B.Sc. CSIT

Year II/IV Semester

013BSCIT002

**Submitted To**

Er. Sanjay Kumar Yadav

Lecturer,

Department of Computer Science

St. Xavier’s College

Maitighar, Kathmandu

**Submitted On**

October 30, 2015

**Transaction Management**

**Introduction**

Concurrent execution of user programs is essential for good Database Management System (DBMS) performance. Because disk accesses are frequent and relatively slow, it is important to keep Central Processing Unit (CPU) humming by working on several user programs concurrently.

A user’s program may carry out many operations on the data retrieved from the database, but the Database Management System (DBMS) is only concerned about what data is read / written from / to the database.

A transaction is the DBMS’s abstract view of a user program: a sequence of reads and writes.

**Transactions**

A transaction is an action, or a series of actions, carried out by a single user or an application program, which reads or updates the contents of a database.

A transaction is a ‘logical unit of work’ on a database

* Each transaction does something in the database
* No part of it alone achieves anything of use or interest

Transactions are the unit of recovery, consistency, and integrity as well. They posses following properties,

ACID properties:

* Atomicity
* Consistency
* Isolation
* Durability

A DBMS must ensure four important properties of transactions to maintain data in the face of concurrent access and system failures:

1. Users should be able to regard the execution of each transaction as atomic: Either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).
2. Each transaction, run by itself with no concurrent execution of other transactions, lnust preserve the consistency of the database. The DBMS assumes that consistency holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.
3. Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as isolation: Transactions are isolated, or protected, from the effects of concurrently scheduling other transactions.
4. Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called durability.

The acronym ACID is sometimes used to refer to these four properties of transactions: atomicity, consistency, isolation and durability. We now consider how each of these properties is ensured in a DBMS.

**Transaction Recovery**

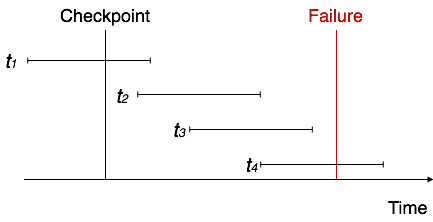
When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −



* The recovery system reads the logs backwards from the end to the last checkpoint.
* It maintains two lists, an undo-list and a redo-list.
* If the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>, it puts the transaction in the redo-list.
* If the recovery system sees a log with <Tn, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

UNDO and REDO: lists of transactions

UNDO = all transactions running at the last checkpoint

REDO = empty

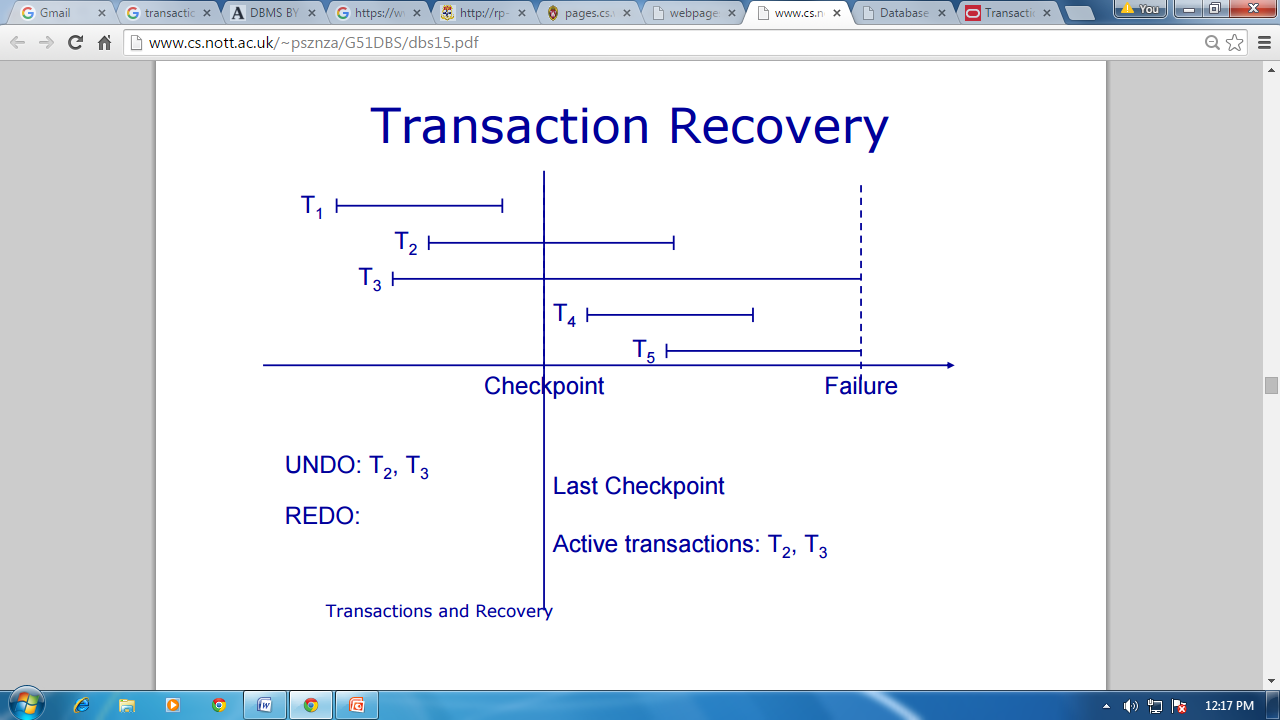
For each entry in the log, starting at the last checkpoint

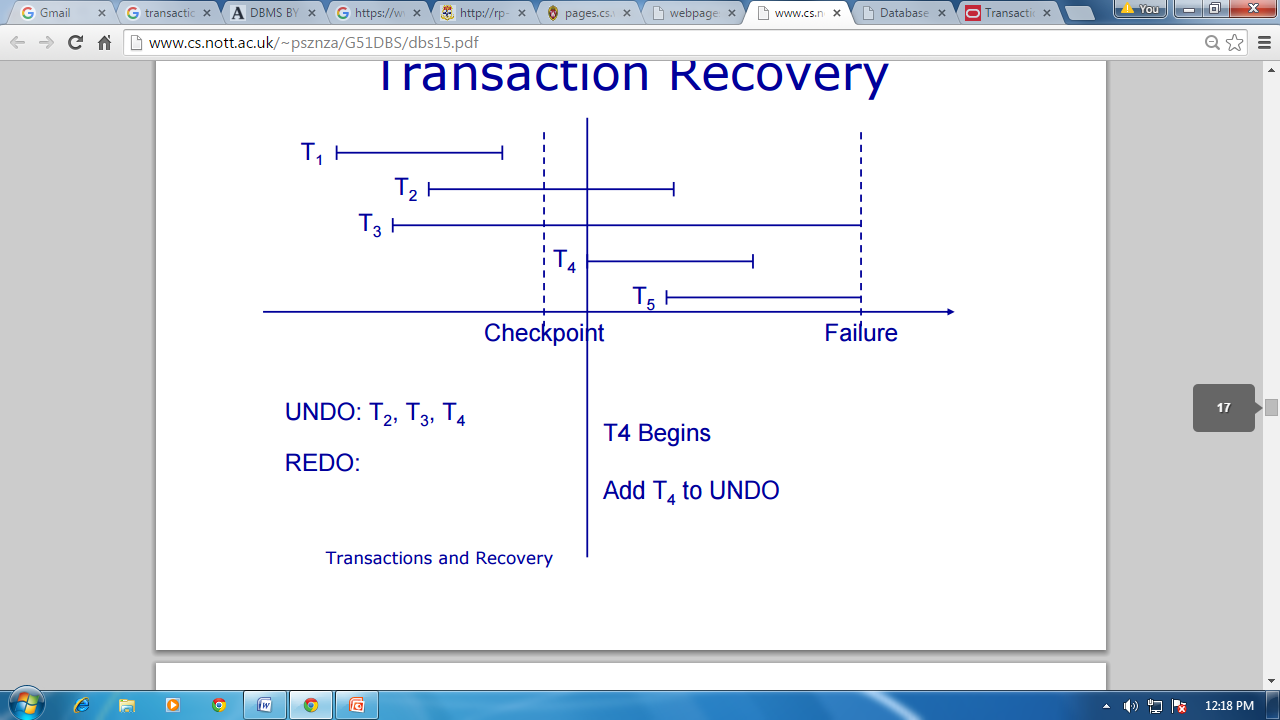
If a BEGIN TRANSACTION entry is found for T

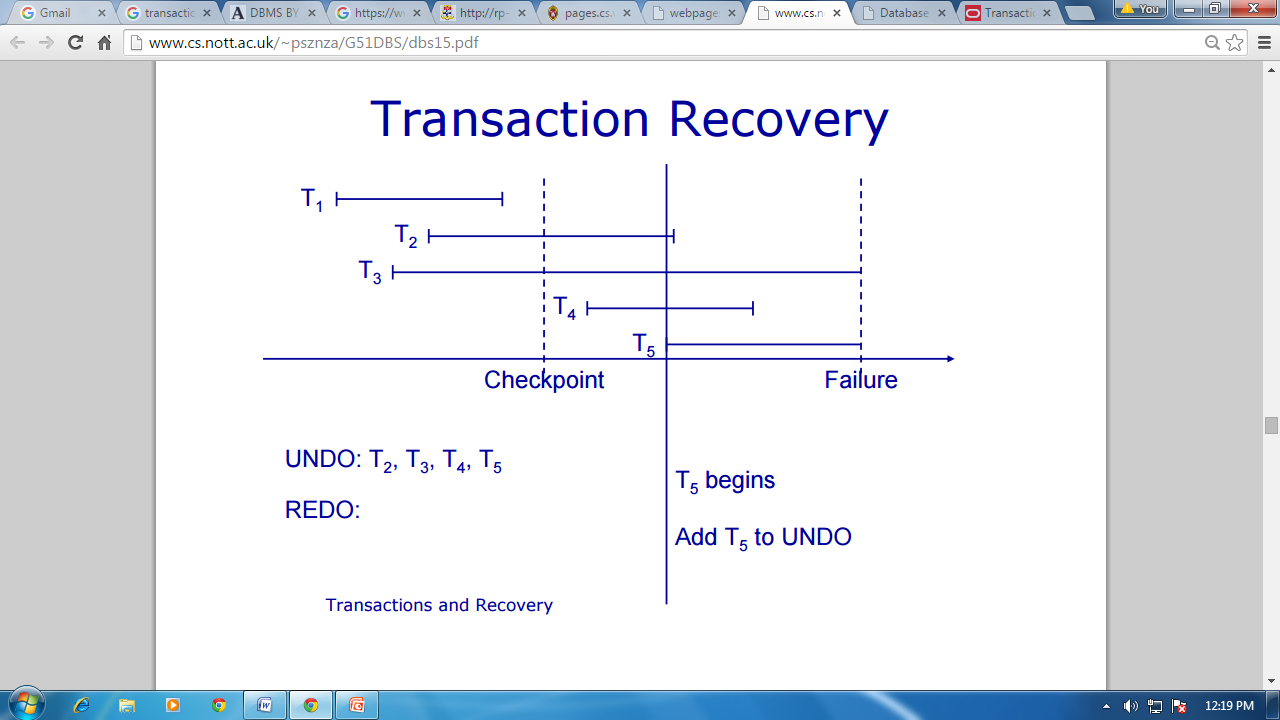
Add T to UNDO

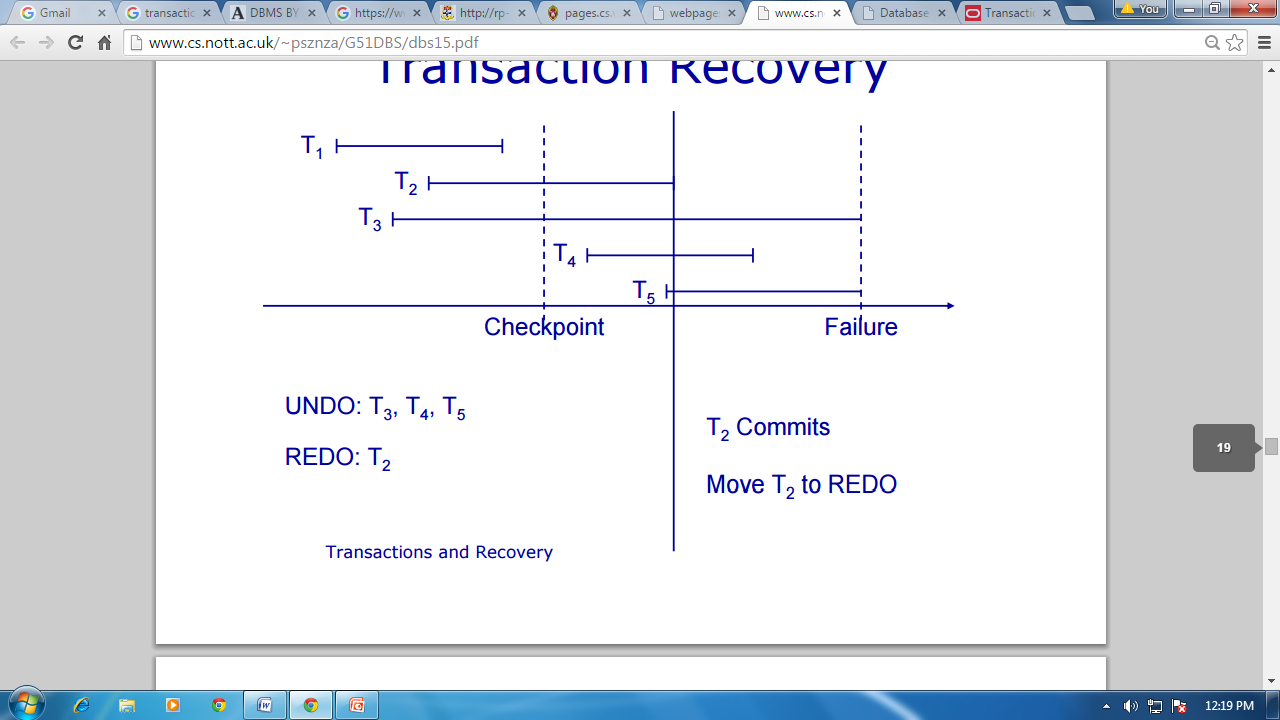
If a COMMIT entry is found for T

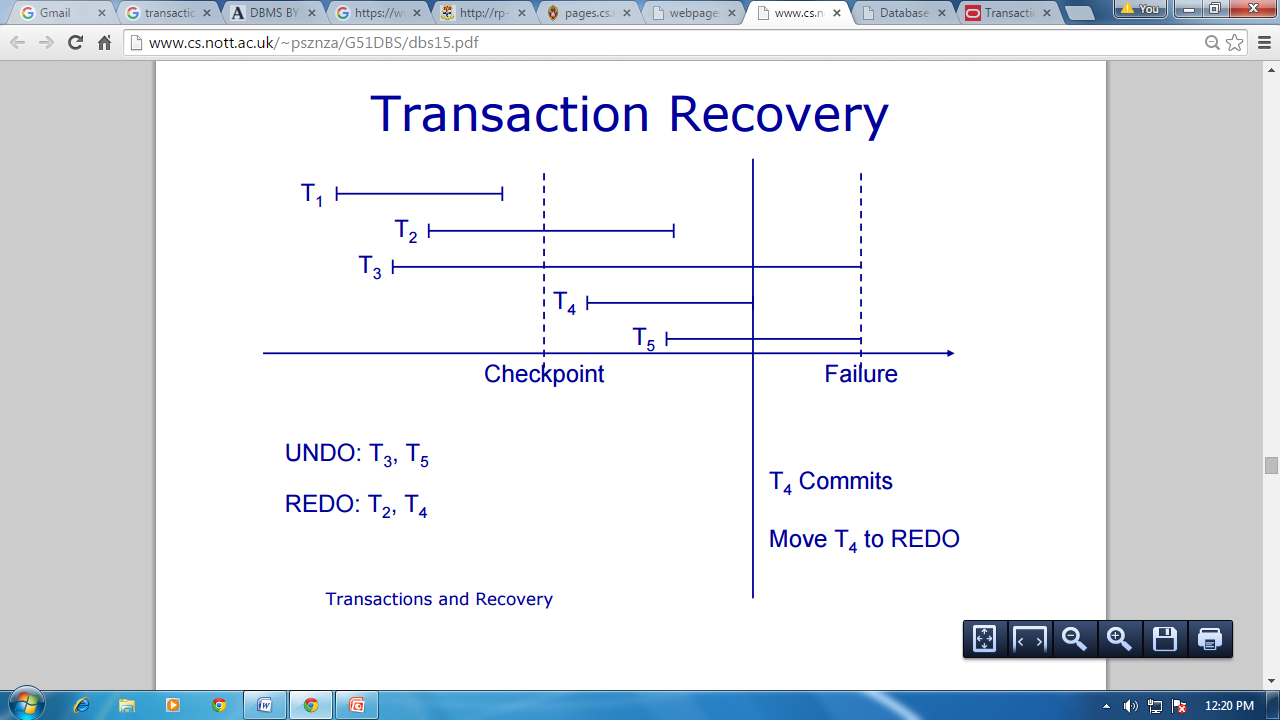
Move T from UNDO to REDO











**System Recovery**

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

* It should check the states of all the transactions, which were being executed.
* A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
* It should check whether the transaction can be completed now or it needs to be rolled back.
* No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

* Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
* Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

**Backwards recovery**

* We need to undo some transactions
* Working backwards through the log we undo any operation by a transaction on the UNDO list
* This returns the database to a consistent state

**Forwards recovery**

* Some transactions need to be redone
* Working forwards through the log we redo any operation by a transaction on the REDO list
* This brings the database up to date

**Media Recovery**

* Restore the database from the last backup
* Use the transaction log to redo any changes made since the last backup
* If the transaction log is damaged you can’t do step 2
* Store the log on a separate physical device to the database
* The risk of losing both is then reduced

**Two-phase Commit**

* Required for distributed or heterogeneous environments, so that correctness is maintained in case of failure during a multi-part COMMIT
* Prepare phase has all local resource managers force logs to a persistent log, local managers reply ok or not
* Commit phase – if all replies are ok, the coordinator commits, and orders the local managers to complete the process; otherwise all are ordered to ROLLBACK

**SQL Facilities**

**Creating and Terminating Transactions:**

A transaction is automatically started when a user executes a statement that accesses either the database or the catalogs, such as a SELECT query, an UPDATE command, or a CREATE TABLE statement.

Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either a COMMIT command or a ROLLBACK (the SQL keyword for abort) command.

In SQL: 1999, two new features are provided to support applications that involve long-running transactions, or that must run several transactions one after the other. To understand these extensions, recall that all the actions of a given transaction are executed in order, regardless of how the actions of different transactions are interleaved. We can think of each transaction as a sequence of steps.

The first feature, called a save point, allows us to identify a point in a transaction and selectively roll back operations carried out after this point. This is especially useful if the transaction carries out what-if kinds of operations, and wishes to undo or keep the changes based on the results. This can be accomplished by defining save points.

In a long-running transaction, we may want to define a series of save points. The save point command allows us to give each save point a name:

SAVEPDINT (save point name)

A subsequent rollback command can specify the save point to roll back to

ROLLBACK TO SAVEPDINT (save point name)

If we define three save points A, B, and C in that order, and then rollback to A, all operations since A are undone, including the creation of save points B and C. Indeed, the save point A is itself undone when we roll hack to it, and we must re-establish it (through another save point Cournand) if we wish to be able to roll back to it again. From a locking standpoint, locks obtained after save point A can be released when we roll back to A.

It is instructive to compare the use of save points with the alternative of executing a series of transactions (i.e., treat all operations in between two consecutive save points as a new transaction). The save point mechanism offers two advantages. First, we can roll back over several save points. In the alternative approach, we can roll back only the most recent transaction, which is equivalent to rolling back to the most recent save point; second, the overhead of initiating several transactions is avoided.

Even with the use of save points, certain applications might require us to run several transactions one after the other. To minimize the overhead in such situations, SQL:1999 introduces another feature, called chained transactions, We can commit or roll back a transaction and immediately initiate another transaction, This is done by using the optional keywords AND CHAIN in the COMMIT and ROLLBACK statements.

**What Should We Lock?**

Until now, we have discussed transactions and concurrency control in tenus of an abstract model in which a database contains a fixed collection of objects, and each transaction is a series of read and write operations on individual objects. An important question to consider in the context of SQL is what the DBMS should treat as an object when setting locks for a given SQL statement (that is part of a transaction).

Consider the following query:

SELECT S.rating, MIN (S.age)

FROM Sailors S

WHERE S.rating = 8

Suppose that this query runs as part of transaction T1 and an SQL statement that modifies the age of a given sailor, say Joe, with rating=8 runs a-s part of transaction T2. What 'objects' should the DBMS lock when executing these transactions? Intuitively, we must detect a conflict between these transactions.

The DBMS could set a shared lock on the entire Sailors table for T1 and set an exclusive lock on Sailors for T2, which would ensure that the two transactions are executed in a serializable manner. However, this approach yields low concurrency, and we can do better by locking smaller objects, reflecting what each transaction actually accesses. Thus, the DBMS could set a shared lock on every row with rating=8 for transaction T1 and set an exclusive lock on just the row for the modified tuple for transaction T2. Now, other read-only transactions that do not involve rating=8 rows can proceed without waiting for T1 or T2.

As this example illustrates, the DBMS can lock objects at different granularities: We can lock entire tables or set row-level locks. The latter approach is taken in current systems because it offers much better performance. In practice, while row-level locking is generally better, the choice of locking granularity is complicated. For example, a transaction that examines several rows and modifies those tha1 satisfy some condition might be best served by setting shared locks on the entire table and setting exclusive locks on those rows it wants to modify. A second point to note is that SQL statements conceptually access a collection of rows described by a .selection predicate. In the preceding example, transaction T1 accesses all rows with rating=8. We suggested that this could be dealt with by setting shared locks on all rows in Sailors that had rating=8. Unfortunately, this is a little too simplistic. To sec why, consider an SQL statement that inserts a new sailor with rating=8 and runs as transaction T3. (Observe that this example violates our assumption of a fixed number of objects in the database, but we must obviously deal with such situations in practice.)

Suppose that the DBMS sets shared locks on every existing Sailors row with rating=8 for Tl. This does not prevent transaction T3 from creating a brand new row with rating=8 and setting an exclusive lock on this row. If this new row has a smaller age value than existing rows, Tl returns an answer that depends on when it executed relative to T2. However, our locking scheme imposes no relative order on these two transactions.

This phenomenon is called the phantom problem: A transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself. To prevent phantoms, the DBMS must conceptually lock all possible rows with rating=8 on behalf of Tl. One way to do this is to lock the entire table, at the cost of low concurrency. It is possible to take advantage of indexes to do better, but in general preventing phantoms can have a significant impact on concurrency.

It may well be that the application invoking T1 can accept the potential inaccuracy due to phantoms. If so, the approach of setting shared locks on existing tuples for Tl is adequate, and offers better performance. SQL allows a programmer to make this choice---and other similar choices'--explicitly, as we see next.

**Transaction Characteristics in SQL**

In order to give programmers control over the locking overhead incurred by their transactions, SQL allows them to specify three characteristics of a transaction: access mode, diagnostics size, and isolation level. The diagnostics size determines the number of error conditions that can be recorded; we will not discuss this feature further. If the access mode is READ ONLY, the transaction is not allowed to modify the database. Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE. For transactions with READ ONLY access mode only shared locks need to be obtained, thereby increasing concurrency.

The isolation level controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The effect of these levels is summarized in Figure IV. In this context, dirty read and unrepeatable read are defined as usual

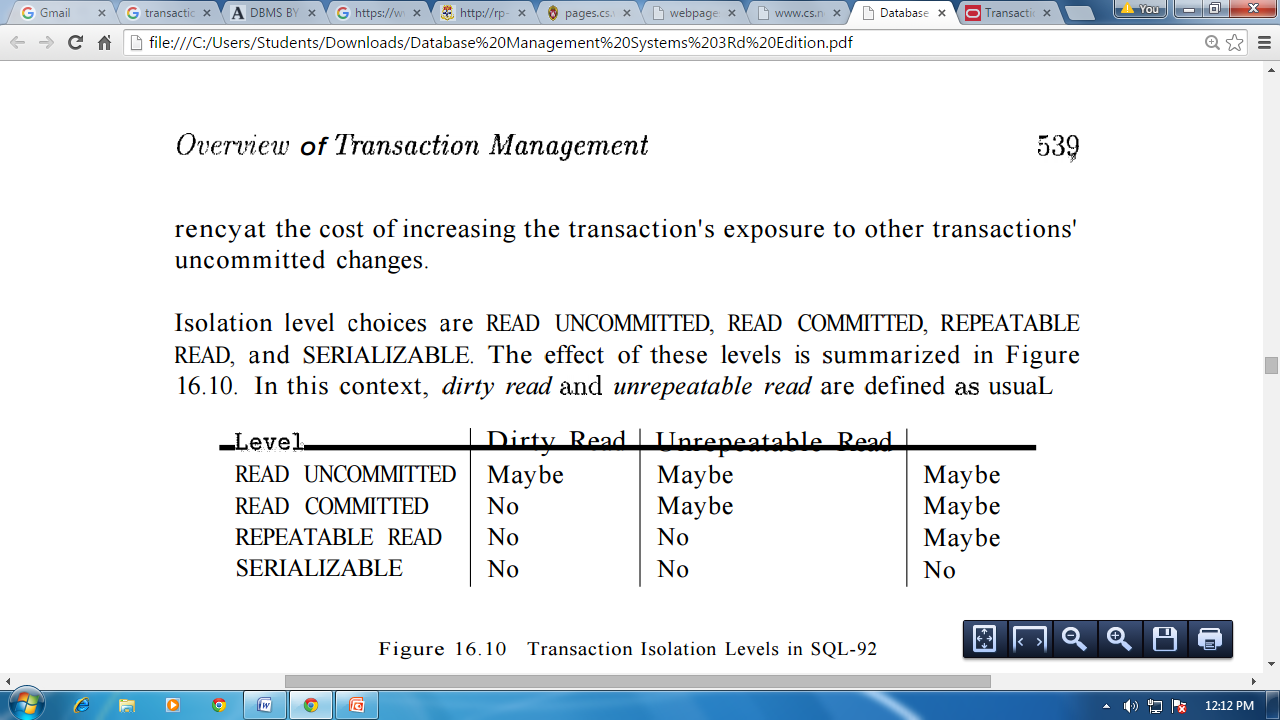


Figure IV: Transaction Isolation Levels in SQL-92

The highest degree of isolation from the effects of other transactions is achieved by setting the isolation level for a transaction T to SERIALIZABLE. This isolation level ensures that T reads only the changes made by committed transactions, that no value read or written by T is changed by any other transaction until T is complete, and that if T reads a set of values based on some search condition, this set is not changed by other transactions until T is complete (i.e., T avoids the phantom phenomenon).

In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires being unchanged and holds them until the end, according to Strict 2PL.

REPEATABLE READ ensures that T reads only the changes made by committed transactions and no value read or written by T is changed by any other transaction until T is complete. However, T could experience the phantom phenomenon; for example, while T examines all Sailors records with rating=1, another transaction might add a new such Sailors record, which is missed by T.

A REPEATABLE READ transaction sets the same locks as a SERIALIZABLE transaction, except that it does not do index locking; that is, it locks only individual objects, not sets of objects.

READ COMMITTED ensures that T reads only the changes made by committed transactions, and that no value written by T is changed by any other transaction until T is complete. However, a value read by T may well be modified by another transaction while T is still in progress, and T is exposed to the phantom problem.

A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that every SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A READ UNCOMMITTED transaction T can read changes made to an object by an ongoing transaction; obviously, the object can be changed further while T is in progress, and T is also vulnerable to the phantom problem.

A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself-a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests.

The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance. For example, a statistical query that finds the average sailor age can be run at the READ COMMITTED level or even the READ UNCOMMITTED level, because a few incorrect or missing values do not significantly affect the result if the number of sailors is large.

The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY

When a transaction is started, the default is SERIALIZABLE and READ WRITE.