SEMINAR REPORT ON

# Transaction Isolation Levels, Implementation of Isolation Levels

Submitted in partial fulfillment of 3rd Semester

MASTER OF COMPUTER APPLICATIONS

Of

Visvesvaraya Technological University

**Md. Miraj Ansari**

Under the Guidance of

**Prof. KALPANA .K**


**Acharya Institute of Technology**

Acharya Dr.Sarvepalli Radhakrishna Road,
Soldevanahalli, Bangalore – 560107
www.acharya.ac.in

2019

ACHARYA INSTITUTE OF TECHNOLOGY
Department of MCA
Soladevanahalli, Hesaraghatta Main Road,
Bengaluru 560107


# C E R T I F I C A T E

This is to certify that the Seminar entitled

**Transaction Isolation Levels,**
**Implementation of Isolation Levels**

is a bonafide work carried out by

## Md. Miraj Ansari

in partial fulfillment of 3rd semester Master of Computer Applications

during the academic year 2019-20.

**Faculty incharge**
**Prof. KALPANA .K**

# INDEX

# Transaction Isolation Levels

Serializability is a useful concept because it allows programmers to ignore issues related to concurrency when they code transactions. If every transaction has the property that it maintains database consistency if executed alone, then serializability ensures that concurrent executions maintain consistency. However, the protocols required to ensure serializability may allow too little concurrency for certain applications. In these cases, weaker levels of consistency are used. The use of weaker levels of consistency places additional burdens on programmers for ensuring database correctness.

The SQL standard also allows a transaction to specify that it may be executed in such a way that it becomes non-serializable with respect to other transactions. For instance, a transaction may operate at the isolation level of **read uncommitted**, which permits the transaction to read a data item even if it was written by a transaction that has not been committed. SQL provides such features for the benefit of long transactions whose results do not need to be precise. If these transactions were to execute in a serializable fashion, they could interfere with other transactions, causing the others' execution to be delayed. The isolation levels specified by the SQL standard are as follows:

• **Serializable** usually ensures serializable execution. However, as we shall explain shortly, some database systems implement this isolation level in a manner that may, in certain cases, allow non-serializable executions.

• **Repeatable read** allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it. However, the transaction may not be serializable

with respect to other transactions. For instance, when it is searching for data satisfying some conditions, a transaction may find some of the data inserted by a committed transaction, but may not find other data inserted by the same transaction.

• **Read committed** allows only committed data to be read, but does not require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.

• **Read uncommitted** allows uncommitted data to be read. It is the lowest isolation level allowed by SQL. All the isolation levels above additionally disallow **dirty writes**, that is, they Disallow writes to a data item that has already been written by another transaction that has not yet committed or aborted. Many database systems run, by default, at the read-committed isolation level. In SQL, it is possible to set the isolation level explicitly, rather than accepting the system's default setting. For example, the statement "**set**

**transaction isolation level serializable;**" sets the isolation level to serializable; any of the other isolation levels may be specified instead. The above syntax is supported by Oracle, PostgreSQL and SQL Server; DB2 uses the syntax "**change isolation level**," with its own abbreviations for isolation levels. Changing of the isolation level must be done as the first statement of a transaction. Further, automatic commit of individual statements must be turned off, if it is on by default; API functions, such as the JDBC method Connection. setAutoCommit(false), can be used to do so. Further, in JDBC the method Connection .setTransactionIsolation(int level) can be used to set the isolation level; see the JDBC manuals for details.

An application designer may decide to accept a weaker isolation level in order to improve system performance. ensuring serializability may force a transaction to wait for other transactions or,

in some cases, to abort because the transaction can no longer be executed as part of a serializable execution. While it may seem short sighted to risk database consistency for performance, this trade-off makes sense if we can be sure that the inconsistency that may occur is not relevant to the application. There are many means of implementing isolation levels. As long as the implementation ensures serializability, the designer of a database application or a user of an application does not need to know the details of such implementations, except perhaps for dealing with performance issues. Unfortunately, even if the

isolation level is set to **serializable**, some database systems actually implement a weaker level of isolation, which does not rule out every possible non-serializable execution; we revisit this issue in Section 14.9. If weaker levels of isolation are used, either explicitly or implicitly, the application designer has to be aware of some details of the implementation, to avoid or minimize the chance of inconsistency due to lack of serializability.

# Implementation of Isolation Levels

Database in a consistent state and allow transaction failures to be handled in a safe manner.

There are various **concurrency-control** policies that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated, regardless of how the operating system time-shares resources (such as CPU time) among the transactions. As a trivial example of a concurrency-control policy, consider this: A transaction acquires a **lock** on the entire database before it starts and releases the lock after it has committed. While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released. As a result of the locking policy, only one transaction can execute at a time. Therefore, only serial schedules are generated. These are trivially serializable, and it is easy to verify that they are recoverable and cascadeless as well.

A concurrency-control policy such as this one leads to poor performance, since it forces transactions to wait for preceding transactions to finish before they can start. In other words, it provides a poor degree of concurrency (indeed, no concurrency at all). The goal of concurrency-control policies is to provide a high degree of concurrency, while ensuring that all schedules that can be generated are conflict or view serializable, recoverable, and cascadeless.

## Locking

Instead of locking the entire database, a transaction could, instead, lock only those data items that it accesses. Under such a policy, the transaction must hold locks long enough to ensure serializability, but for a period short enough not to harm performance excessively.

we present the two-phase locking protocol, a simple, widely used technique that ensures serializability. Stated simply, two-phase locking requires a transaction to have two phases, one where it acquires locks but does not release any, and a second phase where the transaction releases locks but does not acquire any. (In practice, locks are usually released only when the transaction completes its execution and has been either committed or aborted.)

Further improvements to locking result if we have two kinds of locks: shared and exclusive. Shared locks are used for data that the transaction reads and exclusive locks are used for those it writes. Many transactions can hold shared locks on the same data item at the same time, but a transaction is allowed an exclusive lock on a data item only if no other transaction holds any lock (regardless of whether shared or exclusive) on the data item. This use of two modes of locks along with two-phase locking allows concurrent reading of data while still ensuring serializability.

## Timestamps

Another category of techniques for the implementation of isolation assigns each transaction a **timestamp**, typically when it begins. For each data item, the system keeps two timestamps. The read timestamp of a data item holds the largest (that is, the most recent) timestamp of those transactions that read the data item. The write timestamp of a data item holds the timestamp of the transaction that wrote the current value of the data item. Timestamps are used to ensure that transactions access each data item in order of the transactions' timestamps if their accesses conflict. When this is not possible, offending transactions are aborted and restarted with a new timestamp.

## Multiple Versions and Snapshot Isolation

By maintaining more than one version of a data item, it is possible to allow a transaction to read an old version of a data item rather than a newer version written by an uncommitted transaction or by a transaction that should come later in the serialization order. There are a variety of multi version concurrency control techniques. One in particular, called **snapshot isolation**, is widely used in practice.

In snapshot isolation, we can imagine that each transaction is given its own version, or snapshot, of the database when it begins.4 It reads data from this private version and is thus isolated from the updates made by other transactions.

If the transaction updates the database, that update appears only in its own version, not in the actual database itself. Information about these updates is saved so that the updates can be applied to the "real" database if the transaction commits.

When a transaction *T* enters the partially committed state, it then proceeds to the committed state only if no other concurrent transaction has modified data that *T* intends to update. Transactions that, as a result, cannot commit abort instead.

Snapshot isolation ensures that attempts to read data never need to wait (unlike locking). Read-only transactions cannot be aborted; only those that modify data run a slight risk of aborting. Since each transaction reads its own version or snapshot of the database, reading data does not cause subsequent update attempts by other transactions to wait (unlike locking). Since most transactions are read-only (and most others read more data than they update), this is often a major source of performance improvement as compared to locking.

The problem with snapshot isolation is that, paradoxically, it provides *too much* isolation. Consider two transactions *T* and *T*_. In a serializable execution, either *T* sees all the updates made by *T*_ or *T*_ sees all the updates made by *T*, because one must follow the other in the serialization order. Under snapshot isolation, there are cases where neither transaction sees the updates of the other. This is a situation that cannot occur in a serializable execution. In many (indeed, most) cases, the data accesses by the two transactions do not conflict and there is no problem. However, if *T* reads some data item that *T*_ updates and *T*_ reads some data item that *T* updates, it is possible that both transactions fail to read
the update made by the other.

Oracle, PostgreSQL, and SQL Server offer the option of snapshot isolation. Oracle and PostgreSQL implement the **serializable** isolation level using snapshot isolation. As a result, their implementation of serializability can, in exceptional circumstances, result in a non-serializable execution being allowed. SQL Server instead includes an additional isolation level beyond the standard ones, called **snapshot**, to offer the option of snapshot isolation.

## Transactions as SQL Statements

we presented the SQL syntax for specifying the beginning and end of transactions. Now that we have seen some of the issues in ensuring the ACID properties for transactions, we are ready to consider how those properties are ensured when transactions are specified as a sequence of SQL statements rather than the restricted model of simple reads and writes that we considered up to this point.

In our simple model, we assumed a set of data items exists. While our simple model allowed data-item values to be changed, it did not allow data items to be created or deleted. In SQL, however, **insert** statements create new data and **delete** statements delete data. These two statements are, in effect, **write** operations, since they change the database, but their interactions with the actions of other transactions are different from what we saw in our simple model. As an example, consider the following SQL query on our university database that finds all instructors who earn more than $90,000.

**select** *ID*, *name*
**from** *instructor*
**where** *salary* > 90000;

Using our sample *instructor* relation , we find that only Einstein and Brandt satisfy the condition. Now assume that around the same time we are running our query, another user inserts a new instructor named "James" whose salary is $100,000.

**insert into** *instructor* **values** ('11111', 'James', 'Marketing', 100000);

The result of our query will be different depending on whether this insert comes before or after our query is run. In a concurrent execution of these transactions, it is intuitively clear that they conflict, but this is a conflict not captured by our simple model. This situation is referred to as the **phantom phenomenon**, because a conflict may exist on "phantom" data. Our simple model of transactions required that operations operate on a specific data item given as an argument to the operation. In our simple model, we can look at the **read** and **write** steps to see which data items are referenced. But in an SQL statement, the specific data items (tuples) referenced may be determined by a **where** clause predicate. So the same transaction, if run more than once, might reference different data items each time it is run if the values in the database change between runs.

One way of dealing with the above problem is to recognize that it is not sufficient for concurrency control to consider only the tuples that are accessed by a transaction; the information used to find the tuples that are accessed by the transaction must also be considered for the purpose of concurrency control. The information used to find tuples could be updated by an insertion or deletion, or in the case of an index, even by an update to a search-key attribute. For example, if locking is used for concurrency control, the data structures that track the tuples in a relation, as well as index structures, must be appropriately locked. However, such locking can lead to poor concurrency in some situations; index-locking protocols which maximize concurrency, while ensuring serializability in spite of inserts, deletes, and predicates in queries Let us consider again the query:

**select** *ID*, *name*
**from** *instructor*
**where** *salary*> 90000;

and the following SQL update:

**update** *instructor*
**set** *salary* = *salary* * 0.9
**where** *name* = 'Wu';

We now face an interesting situation in determining whether our query conflicts with the update statement. If our query reads the entire *instructor* relation, then it reads the tuple with Wu's data and conflicts with the update. However, if an index were available that allowed our query direct access to those tuples with *salary*

> 90000, then our query would not have accessed Wu's data at all because Wu's salary is initially $90,000 in our example instructor relation, and reduces to $81,000 after the update.

However, using the above approach, it would appear that the existence of a conflict depends on a low-level query processing decision by the system that is unrelated to a user-level view of the meaning of the two SQL statements! An alternative approach to concurrency control treats an insert, delete or update as conflicting with a predicate on a relation, if it could affect the set of tuples selected by a predicate. In our example query above, the predicate is "*salary* > 90000", and an update of Wu's salary from $90,000 to a value greater than $90,000, or an update of Einstein's salary from a value greater that $90,000 to a value less than or equal to $90,000, would conflict with this predicate. Locking based on this idea is called **predicate locking**; however predicate locking is expensive, and not used in practice.