四川大學
SICHUAN UNIVERSITY

Database System Concepts

Transactions

伍元凯

College of Computer Science (Software), Sichuan University

wuyk0@scu.edu.cn

2023/05/16

海纳百川
有容乃大

A transaction is a **unit** of program execution that accesses and possibly updates various data items.

- SQL
- C++ or Java
- JDBC or ODBC

Typical transactions:

```
EGIN TRANSACTION;

-- SQL statements here

COMMIT;
```

ACID 酸

Atomicity（原子性）

Either all operations of the transaction are reflected properly in the database, or none are.

Consistency（一致性）

Execution of a transaction in isolation (i.e., with no other transaction executing concurrently) preserves the consistency of the database.

ACID 酸

Isolation（隔离性）

Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished.

Ensuring the isolation property may have a significant adverse effect on system performance.

Durability（持久性）

After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

目录

海纳百川 有容乃大

例 1

Transfers $50 from account A to account B.

$$\begin{aligned} T_i : read(A); \\ A := A - 50; \\ write(A); \\ read(B); \\ B := B + 50; \\ write(B). \end{aligned}$$

Atomicity（原子性）

The values of accounts A and B are \$1000 and \$2000.
Suppose that a failure happened after the $write(A)$ operation but before the $write(B)$ operation.
The values of accounts A and B are **\$950 and \$2000**.
We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction.

Consistency（一致性）

The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction

Isolation（隔离性）

The database is temporarily **inconsistent** while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes $A + B$, it will observe an inconsistent value. Ensuring the isolation property is the responsibility of the concurrency-control system

Durability（持久性）

The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

Volatile storage

Computer memory that requires **power** to maintain the stored information; it retains its contents while powered on but when the power is interrupted, the stored data is **quickly lost**.
Random-access memory (RAM; **内存条**) is a form of computer memory that can be read and changed in any order。

Non-volatile storage

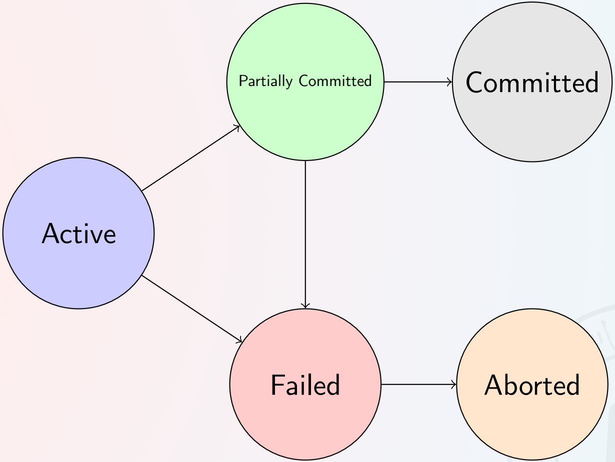
Information residing in non-volatile storage survives system crashes.
Examples of non-volatile storage include secondary storage devices such as magnetic disk and flash storage,

A black hole may envelop the earth and permanently destroy all data!

To implement stable storage, we replicate the information in several non-volatile storage media (usually disk) with independent failure modes. The degree to which a system ensures durability and atomicity depends on how stable its implementation of stable storage really is.

A transaction must be in one of the following states:

- Active, the initial state; the transaction stays in this state while it is executing.
- Partially committed, after the final statement has been executed.
- Failed, after the discovery that normal execution can no longer proceed.
- Aborted, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- Committed, after successful completion.



Transaction States

The system has two options if it enters the aborted state:

- It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the **internal logic of the transaction**. A restarted transaction is considered to be a new transaction.
- It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

例 2

Be careful with **observable external writes**

consider a user making a booking over the web. It is possible that the database system or the application server crashes just after the booking transaction commits. It is also possible that the network connection to the user is lost just after the booking transaction commits. In either case, even though the transaction has committed, the external write has not taken place. To handle such situations, the application must be designed such that when the user connects to the web application again, the user will be able to see whether her transaction had succeeded or not.

The easiest way

Transactions run serially—that is, one at a time, each starting only after the previous one has completed.

Reasons for allowing concurrency:

- Improved throughput and resource utilization.
- Reduced waiting time.

Transaction T_1 transfers \$50 from account A to account B.

$$
\begin{aligned}
read(A);\\
A := A - 50;\\
write(A);\\
read(B);\\
B := B + 50;\\
write(B).
\end{aligned}
$$

Transaction T_2 transfers 10 percent of the balance from account A to account B.

$$
\begin{aligned}
read(A);\\
temp := A * 0.1;\\
A := A - temp;\\
write(A);\\
read(B);\\
B := B + temp;\\
write(B).
\end{aligned}
$$

Serial schedule

T_1	T_2
$read(A)$	
$A := A - 50$	
$write(A)$	
$read(B)$	
$B := B + 50$	
$write(B)$	
commit	
	$read(A)$
	$temp := A * 0.1$
	$A := A - temp$
	$write(A)$
	$read(B)$
	$B := B + temp$
	$write(B)$
	commit

The execution sequences are called schedules. They represent the chronological order in which instructions are executed in the system.

Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.
For a set of n transactions, there exist n factorial ($n!$) different valid serial schedules.

If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

Concurrent schedule

T_1	T_2
$read(A)$ $A := A - 50$ $write(A)$	
	$read(A)$ $temp := A * 0.1$ $A := A - temp$ $write(A)$
$read(B)$ $B := B + 50$ $write(B)$ commit	
	$read(B)$ $B := B + temp$ $write(B)$ commit

Incorrect schedule

T_1	T_2
$read(A)$	
$A := A - 50$	
	$read(A)$
	$temp := A * 0.1$
	$A := A - temp$
	$write(A)$
$write(A)$	
$read(B)$	
$B := B + 50$	
$write(B)$	
commit	
	$read(B)$
	$B := B + temp$
	$write(B)$
	commit

It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The concurrency-control（并发控制） component of the database system carries out this task.

the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called serializable（序列化） schedules.

Conflict serializability (consider only two operations: read and write)

Let us consider a schedule S in which there are two consecutive instructions, I and J, of transactions T_i and T_j, respectively ($i \neq j$). If I and J refer to different data items, then we can swap I and J without affecting the results of any instruction in the schedule. However, if I and J refer to the same data item Q, then the order of the two steps may matter.

1. $I = \text{read}(Q)$, $J = \text{read}(Q)$. The order of I and J does not matter, since the same value of Q is read by T_i and T_j, regardless of the order.
2. $I = \text{read}(Q)$, $J = \text{write}(Q)$. If I comes before J, then T_i does not read the value of Q that is written by T_j in instruction J. If J comes before I, then T_i reads the value of Q that is written by T_j. Thus, the order of I and J matters.
3. $I = \text{write}(Q)$, $J = \text{read}(Q)$. The order of I and J matters for reasons similar to those of the previous case.
4. $I = \text{write}(Q)$, $J = \text{write}(Q)$. Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j. However, the value obtained by the next $\text{read}(Q)$ instruction of S is affected. If there is no other $\text{write}(Q)$ instruction after I and J in S, then the order of I and J directly affects the final value of Q in the database state that results from schedule S.

We say that I and J conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

T_3	T_4
$read(A)$	
$wite(A)$	
	$read(A)$
	$wite(A)$
$read(B)$	
$write(B)$	
	$read(B)$
	$write(B)$

Let I and J be consecutive instructions of a schedule S. If I and J are instructions of different transactions and I and J do not conflict, then we can swap the order of I and J to produce a new schedule S'. S is equivalent to S', since all instructions appear in the same order in both schedules except for I and J, whose order does not matter.

Since the $write(A)$ instruction of T_2 does not conflict with the $read(B)$ instruction of T_1

T_3	T_4
$read(A)$	
$wite(A)$	
	$read(A)$
$read(B)$	
	$wite(A)$
$write(B)$	
	$read(B)$
	$write(B)$

1. Swap the `read(B)` instruction of T_1 with the `read(A)` instruction of T_2.
2. Swap the `write(B)` instruction of T_1 with the `write(A)` instruction of T_2.
3. Swap the `write(B)` instruction of T_1 with the `read(A)` instruction of T_2.

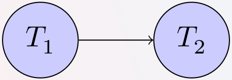
T_3	T_4
$read(A)$	
$wite(A)$	
$read(B)$	
$write(B)$	
	$read(A)$
	$wite(A)$
	$read(B)$
	$write(B)$

If a schedule S can be transformed into a schedule S' by a series of swaps of nonconflicting instructions, we say that S and S' are conflict equivalent.

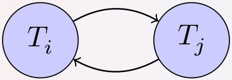
We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

Consider a schedule S. We construct a directed graph, called a precedence graph, from S. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \to T_j$ for which one of three conditions holds:

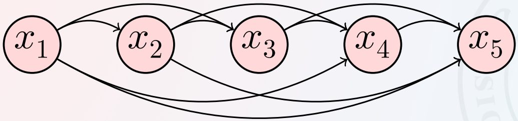
1. T_i executes $write(Q)$ before T_j executes $read(Q)$.
2. T_i executes $read(Q)$ before T_j executes $write(Q)$.
3. T_i executes $write(Q)$ before T_j executes $write(Q)$.



Precedence graph for the first example

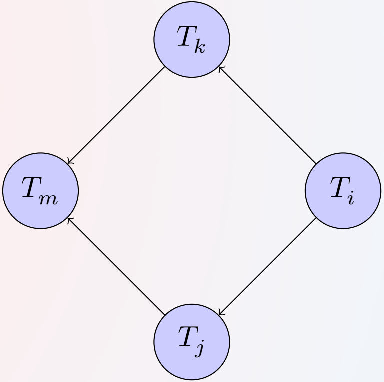


If the precedence graph for S has a cycle, then schedule S is not conflict serializable



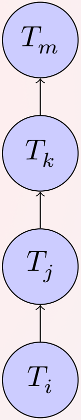
Complex one

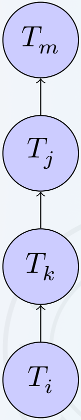
A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called topological sorting.



A precedence graph

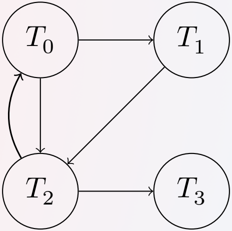
Illustration of topological sorting



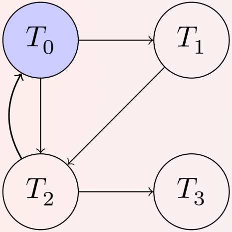


Cycle-detection algorithms

Contains a cycle, indicating that this schedule is not conflict serializable.

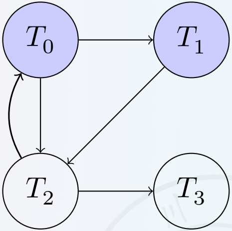


Initially, 0 will be marked in both the visited[] and recStack[] array as it is a part of the current path.



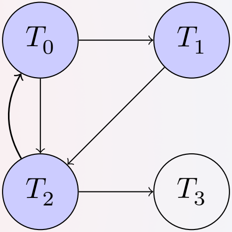
	0	1	2	3
visited	true	false	false	false
recStack	true	false	false	false

Now 0 has two adjacent vertices 1 and 2. Let us consider traversal to the vertex 1.



	0	1	2	3
visited	true	true	false	false
recStack	true	true	false	false

Vertex 1 has only one adjacent vertex. Call the recursive function for 2 and mark it in visited[] and recStack[].



	0	1	2	3
visited	true	true	true	false
recStack	true	true	true	false

Vertex 2 also has two adjacent vertices.
Vertex 0 is visited and already marked in the recStack[]. So if 0 is checked first, we will get the answer that there is a cycle present.

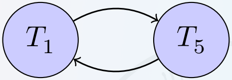
Depth First Traversal (DFS):

```python
class Graph():
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def isCyclicUtil(self, v, visited, recStack):
        # Mark current node as visited and
        # adds to recursion stack
        visited[v] = True
        recStack[v] = True
        # Recur for all neighbours
        # if any neighbour is visited and in
        # recStack then graph is cyclic
        for neighbour in self.graph[v]:
            if visited[neighbour] == False:
                if self.isCyclicUtil(neighbour, visited, recStack)
```

```
                    == True:
                    return True
            elif recStack[neighbour] == True:
                return True
        # The node needs to be popped from
        # recursion stack before function ends
        recStack[v] = False
        return False
    # Returns true if graph is cyclic else false
    def isCyclic(self):
        visited = [False] * (self.V + 1)
        recStack = [False] * (self.V + 1)
        for node in range(self.V):
            if visited[node] == False:
                if self.isCyclicUtil(node, visited, recStack) ==
                    True:
                    return True
        return False
```

It is possible to have two schedules that produce the same outcome but that are not conflict equivalent.

T_1	T_5
$read(A)$	
$A := A - 50$	
$wite(A)$	
	$read(B)$
	$B := B - 10$
	$write(B)$
$read(B)$	
$B := B + 50$	
$write(B)$	
	$read(A)$
	$A := A + 10$
	$write(A)$



not conflict equivalent

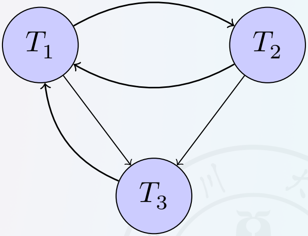
View Equivalent Schedules

Schedules S_1 and S_2 are called view equivalent if:

- For each data item X, if transaction T_i reads X from the database initially in schedule S_1, then in schedule S_2 also, T_i must perform the initial read of X from the database.
- If transaction T_i reads a data item that has been updated by the transaction T_j in schedule S_1, then in schedule S_2 also, transaction T_i must read the same data item that has been updated by the transaction T_j.
- For each data item X, if X has been updated at last by transaction T_i in schedule S_1, then in schedule S_2 also, X must be updated at last by transaction T_i.

Checking view serializability

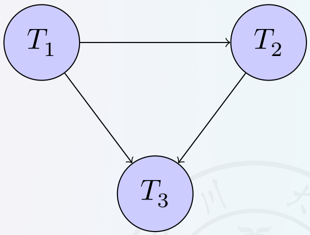
T_1	T_2	T_3
$read(A)$		
	$write(A)$	
		$read(A)$
$write(A)$		
		$write(A)$



not conflict serializable

- There exists a **blind write** T_2 in the given schedule S. Therefore, the given schedule S may or may not be view serializable.

- T1 firstly reads A and T2 firstly updates A.
- So, $T1$ must execute before $T2$.
- Thus, we get the dependency $T1 \rightarrow T2$.
- Final updation on A is made by the transaction $T3$.
- So, $T3$ must execute after all other transactions.
- Thus, we get the dependency $(T1, T2) \rightarrow T3$.
- From write-read sequence, we get the dependency $T2 \rightarrow T3$.



Clearly, there exists **no cycle in the dependency graph**. Therefore, the given schedule S is view serializable.

目录

Partial schedule

Not included a commit or abort operation for T_6

T_7 is dependent on T_6

T_7 commits while T_6 is still in the active state. Now suppose that T_6 fails before it commits. T_7 has read the value of data item A written by T_6.

T_6	T_7
$read(A)$	
$write(A)$	
	$read(A)$
	$commit$
$read(B)$	

Recoverable schedule

For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j.

T_6	T_7
$read(A)$	
$write(A)$	
$commit$	
	$read(A)$
	$commit$
$read(B)$	

目录

T_8	T_9	T_{10}
$read(A)$		
$read(B)$		
$write(A)$		
	$read(A)$	
	$write(A)$	
		$read(A)$
$abort$		

Cascading rollback

T_8 fails. T_8 must be rolled back. Since T_9 is dependent on T_8, T_9 must be rolled back. Since T_{10} is dependent on T_9, T_{10} must be rolled back.

Cascadeless schedule

for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

Serializability ensures that **concurrent** executions maintain **consistency**.

T_1	T_2
$write(A)$	
$Commit$	
	$read(A)$
	$Commit$

T_1

```
BEGIN TRANSACTION;
UPDATE Employees SET Salary = Salary + 1000 WHERE Name = 'John';
COMMIT;
```

T_2

```
BEGIN TRANSACTION;
SELECT Name, Salary FROM Employees WHERE Salary > 50000;
COMMIT;
```

By default, T_2 would execute in a serializable manner, ensuring that it reads a consistent snapshot of the data, even if T_1 is concurrently modifying it.

T_2

```sql
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT Name, Salary FROM Employees WHERE Salary > 50000;
COMMIT;
```

In this case, T_2 explicitly sets the isolation level to "read uncommitted" before executing the SELECT statement. This allows T_2 to read and return the intermediate state of the data modified by T_1, even before T_1 commits.

The isolation levels specified by the SQL standard are as follows:

- Serializable usually ensures serializable execution.
- 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.
- Read committed allows only committed data to be read, but does not require repeatable reads.
- Read uncommitted allows uncommitted data to be read.

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.

Higher level setting

```
set transaction isolation level serializable
```

By default, most databases commit individual statements as soon as they are executed. The command **start transaction** ensures that subsequent SQL statements, until a subsequent commit or rollback, are executed as a single transaction. As expected, the **commit** operation commits the preceding SQL statements, while **rollback** rolls back the preceding SQL statements.

level setting of JDBC

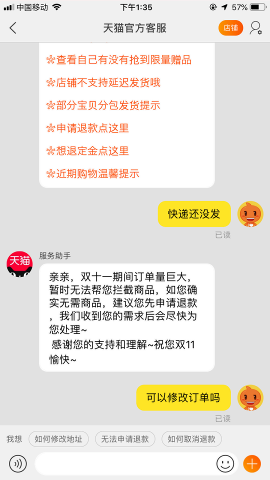
```
mport java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
public class SerializableTransactionExample {
    public static void main(String[] args) {
        // Database connection details
        String jdbcUrl = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "myuser";
        String password = "mypassword";
        // SQL statements for transactions
        String transaction1 = "UPDATE Employees SET Salary = Salary
            + 1000 WHERE Name = 'John'";
        String transaction2 = "SELECT Name, Salary FROM Employees
            WHERE Salary > 50000";
        try (Connection connection = DriverManager.getConnection(
            jdbcUrl, username, password)) {
```

```
        // Set the transaction isolation level to SERIALIZABLE
                    connection.setTransactionIsolation(
            Connection.TRANSACTION_SERIALIZABLE);
        // Start transaction 1
        connection.setAutoCommit(false);
        try (Statement statement = connection.createStatement()
            ) {
            statement.executeUpdate(transaction1);
            connection.commit();
            System.out.println("Transaction 1 committed.");
        } catch (SQLException e) {
            connection.rollback();
            System.out.println("Transaction 1 rolled back.");
        }
        // Start transaction 2
        connection.setAutoCommit(false);
        try (Statement statement = connection.createStatement()
            ) {
            statement.executeQuery(transaction2);
            connection.commit();
            System.out.println("Transaction 2 committed.");
        } catch (SQLException e) {
            connection.rollback();
```

In JDBC the method **setTransactionIsolation(int level)** of the Connection interface can be invoked with any one of

- Connection.TRANSACTION SERIALIZABLE,
- Connection.TRANSACTION REPEATABLE READ,
- Connection.TRANSACTION READ COMMITTED, or
- Connection.TRANSACTION READ UNCOMMITTED

Serializable schedules are the ideal way to ensure consistency, but in our day-today lives, we don’t impose such stringent requirements.



海纳百川 有容乃大

While one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

Two locks mode:

- Shared. If a transaction T_i has obtained a shared-mode lock (denoted by S) on item Q, then T_i can read, but cannot write, Q.
- Exclusive. If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on item Q, then T_i can both read and write Q.

We require that every transaction request a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q.

The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction.

Suppose that a transaction T_i requests a lock of mode A on item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B. If transaction T_i can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is compatible（兼容）with mode B.

To access a data item, transaction T_i must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency-control manager will not grant the lock until all incompatible locks held by other transactions have been released.

	S	X
S	true	false
X	false	false

Lock-compatibility matrix comp

T_1	T_2	concurrency-control manager
$lock-X(B)$		
		$grant-X(B,T_1)$
$read(B)$		
$B := B - 50$		
$write(B)$		
$unlock(B)$		
	$lock-S(A)$	
		$grant-S(A,T_2)$
	$read(A)$	
	$unlock(A)$	
	$lock-S(B)$	
		$grant-S(B,T_2)$
	$read(B)$	
	$unlock(B)$	
	$display(A+B)$	
$lock-X(A)$		
		$grant-X(A,T_1)$
$read(A)$		
$A := A + 50$		
$write(A)$		
$unlock(A)$		

deadlock

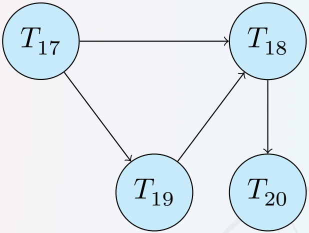
Since T_3 is holding an exclusive-mode lock on B and T_4 is requesting a shared-mode lock on B, T_4 is waiting for T_3 to unlock B. Similarly, since T_4 is holding a shared-mode lock on A and T_3 is requesting an exclusive-mode lock on A, T_3 is waiting for T_4 to unlock A.

T_3	T_4
$lock-X(B)$	
$read(B)$	
$B := B - 50$	
$write(B)$	
	$lock-S(A)$
	$read(A)$
	$lock-S(B)$
$lock-X(A)$	

Deadlocks can be described precisely in terms of a directed graph called a wait-for graph.

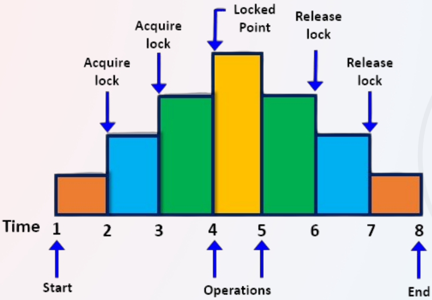
The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair $T_i \rightarrow T_j$. If $T_i \rightarrow T_j$ is in E, then there is a directed edge from transaction T_i to T_j, implying that transaction T_i is waiting for transaction T_j to release a data item that it needs.

- Transaction T_{17} is waiting for transactions T_{18} and T_{19}.
- Transaction T_{19} is waiting for transaction T_{18}.
- Transaction T_{18} is waiting for transaction T_{20}.



One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

- Growing phase. A transaction may obtain locks, but may not release any lock.
- Shrinking phase. A transaction may release locks, but may not obtain any new locks.



T_5	T_6	T_7
$lock-X(A)$		
$read(A)$		
$lock-S(B)$		
$read(B)$		
$write(A)$		
$unlock(A)$		
	$lock-X(A)$	
	$read(A)$	
	$write(A)$	
	$unlock(A)$	
		$lock-S(A)$
		$read(A)$

Partial schedule under two-phase locking.

Cascading rollbacks can be avoided by a modification of two-phase locking called the strict two-phase locking protocol. This protocol requires not only that locking be two phase, but also that all **exclusive-mode locks taken by a transaction be held until that transaction commits**.

Another variant of two-phase locking is the rigorous two-phase locking protocol, which requires that all **locks be held until the transaction commits**.

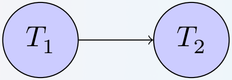
A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction

- When a transaction T_i issues a read(Q) operation, the system issues a lock-S(Q) instruction followed by the read(Q) instruction.
- When T_i issues a write(Q) operation, the system checks to see whether T_i already holds a shared lock on Q. If it does, then the system issues an upgrade(Q) instruction, followed by the write(Q) instruction. Otherwise, the system issues a lock-X(Q) instruction, followed by the write(Q) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

Proof: two-phase locking protocol ensures conflict serializability

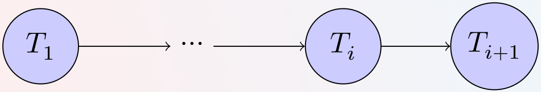
Possibility:

- $Read_1(A) \rightarrow Write_2(A)$
- $Write_1(A) \rightarrow Read_2(A)$
- $Write_1(A) \rightarrow Write_2(A)$



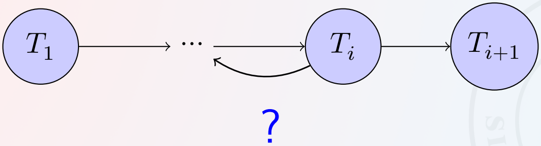
例 3

Summary: T_1 releases lock on $A \rightarrow T_2$ acquires lock on A



例 4

Summary: T_1 releases lock on $A_1 \rightarrow T_2$ acquires lock on $A_1 \rightarrow T_2$ releases lock on $A_2 \cdots \rightarrow T_i$ releases lock on $Z \rightarrow T_{i+1}$ acquires lock on Z



Path T_1 to T_2 means T_1 release lock before T_2 acquires lock

Cycles means T_i releases lock before T_i acquires lock

2PL dose not release lock before acquiring lock

We can not have a cycle

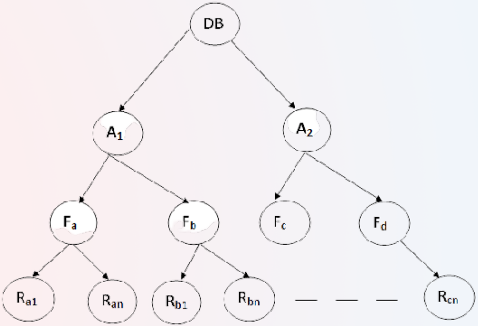


Figure: Multi Granularity tree Hierarchy

- Database: (DB)
- Area: A_i
- File: F_{ij}
- record: r_{ijk}

if transaction T_j needs to access only a few tuples, it should not be required to lock the entire relation, since otherwise concurrency is lost.

Intention Mode Lock:

- **Intention-shared (IS)**: It contains explicit locking at a lower level of the tree but only with shared locks.
- **Intention-Exclusive (IX)**: It contains explicit locking at a lower level with exclusive or shared locks.
- **Shared Intention-Exclusive (SIX)**:the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Lock-compatibility matrix comp

- Transaction T_i must observe the lock-compatibility function.
- Transaction T_i must lock the root of the tree first and can lock it in any mode.
- Transaction T_i can lock a node Q in S or IS mode only if T_i currently has the parent of Q locked in either IX or IS mode.
- Transaction T_i can lock a node Q in X, SIX, or IX mode only if T_i currently has the parent of Q locked in either IX or SIX mode.
- Transaction T_i can lock a node only if T_i has not previously unlocked any node (i.e., T_i is two-phase).
- Transaction T_i can unlock a node Q only if T_i currently has none of the children of Q locked.

Timestamps | 目录

海纳百川 有容乃大

Timestamp

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.

Timestamp control in Java

```java
mport java.util.concurrent.atomic.AtomicInteger;
class Transaction {
    private static final AtomicInteger globalTimestamp = new
        AtomicInteger(0);
    private final int timestamp;
    public Transaction() {
        this.timestamp = globalTimestamp.getAndIncrement();
    }
    public int getTimestamp() {
        return timestamp;
    }
}
public class IsolationWithTimestamps {
    public static void main(String[] args) {
        Transaction t1 = new Transaction();
        Transaction t2 = new Transaction();
        System.out.println("Transaction t1 timestamp: " + t1.
```

```
                getTimestamp());
        System.out.println("Transaction t2 timestamp: " + t2.
                getTimestamp());
    }
}
```

目录

snapshot isolation

Each transaction is given its own version, or snapshot, of the database when it begins. 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.

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 (**update**).

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.

phantom phenomenon 幻影读现象

Transaction1

```sql
select ID, name
from instructor
where salary > 90000;
```

Transaction2

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

The result of our query depends on whether this insert comes before or after our query is run.

Let T denote the query and let T' denote the insert. If T' comes first, then there is an edge $T' \rightarrow T$ in the precedence graph.

Transaction1

```sql
select ID, name
from instructor
where salary> 90000;
```

Transaction2

```sql
update instructor
set salary = salary * 0.9
where name = ’ Wu’ ;
```

If our query reads the entire instructor relation, then it reads the tuple withWu’ s data and conflicts with the update.

海纳百川 有容乃大

Let I_i and I_j be instructions of T_i and T_j, respectively, that appear in schedule S in consecutive order. Let $I_i = \text{delete}(Q)$. We consider several instructions I_j.

- $I_j = \text{read}(Q)$. I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_j can execute the read operation successfully.
- $I_j = \text{write}(Q)$. I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_j can execute the write operation successfully.
- $I_j = \text{delete}(Q)$. I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_i will have a logical error.
- $I_j = \text{insert}(Q)$. I_i and I_j conflict. Suppose that data item Q did not exist prior to the execution of I_i and I_j. Then, if I_i comes before I_j, a logical error results for T_i. If I_j comes before I_i, then no logical error results.

海纳百川 有容乃大

insert(Q) conflicts with a read(Q) operation or a write(Q) operation; no read or write can be performed on a data item before it exists. Since an insert(Q) assigns a value to data item Q, an insert is treated similarly to a write for concurrency-control purposes.

Under the two-phase locking protocol, an exclusive lock is required on a data item before that item can be deleted.

Under the two-phase locking protocol, if T_i performs an insert(Q) operation, T_i is given an exclusive lock on the newly created data item Q.

目录

T_{30}

```
select count(*)
from instructor
where dept name = 'Physics' ;
```

T_{31}

```
insert into instructor
values (11111, 'Feynman', 'Physics', 94000);
```

- If T_{30} uses the tuple newly inserted by T_{31} in computing count$(*)$, then T_{30} reads a value written by T_{31}. Thus, in a serial schedule equivalent to S, T_{31} must come before T_{30}.
- If T_{30} does not use the tuple newly inserted by T_{31} in computing count$(*)$, then in a serial schedule equivalent to S, T_{30} must come before T_{31}.

The index-locking protocol takes advantage of the availability of indices on a relation, by turning instances of the phantom phenomenon into conflicts on locks on index leaf nodes.

- Every relation must have at least one index.
- A transaction Ti can access tuples of a relation only after first finding them through one or more of the indices on the relation.
- A transaction T_i that performs a lookup must acquire a shared lock on all the index leaf nodes that it accesses.
- A transaction T_i may not insert, delete, or update a tuple t_i in a relation r without updating all indices on r. The transaction must obtain exclusive locks on all index leaf nodes that are affected by the insertion, deletion, or update.
- The rules of the two-phase locking protocol must be observed.

Consider the following two transactions:

T_{34}
$read(A)$
$read(B)$
$if \quad A = 0 \quad then \quad B := B + 1;$
$write(B)$

T_{35}
$read(B)$
$read(A)$
$if \quad B = 0 \quad then \quad A := A + 1;$
$write(A)$

Add lock and unlock instructions to transactions T31 and T32 so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

Thanks
End of Chapter 9