Advanced Databases

8. Transaction Management

8. Transaction Management

8.1 Introduction

- DBMS – Overview

Functions
Data Storage, Retrieval, and Update
Provision of a System Catalog
Transaction Support
Concurrency Control Services
Recovery Services
Authorisation Services
Data Communications Support
Integrity Services
Data Independence Support
Utility Services

8.1 Introduction

- Transaction Support
  - What is a *transaction*?

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

  - Examples:
    - Update an student's phone number that is held in a Student_Detail table:
    - Delete an order record from a Order table
    - Create a current account transaction record for an ATM withdrawal in an Account Transaction table **AND** update the associated current balance value that is held in the Account table
  - We must ensure that the database is not left in an <u>inconsistent</u> state

8.1 Introduction

- Concurrency Control Services
  - What is *concurrency control?*

Concurrency control is the process of managing simultaneous operations on the database without having them interfere with one another.

  - Example:

Time	Transaction1	Transaction2
t1	Read A/c 1 Balance	
t2	Add 100 to Balance	Read A/c 1 Balance
t3	Write A/c 1 Balance	Subtract 10 from Balance
t4		Write A/c 1 Balance
t5		

  - We must ensure that the database is updated correctly when multiple users (or application programs) are accessing and updating the database concurrently (i.e., simultaneously).

8.1 Introduction

- Recovery Services
  - What are *recovery services?*

Recovery services provide a set of mechanisms that allow a database to be restored to a correct and consistent state in the event of a failure.

  - Failure examples:
    - System crashes
    - Media failures
    - Application software errors
    - Physical disasters
    - Carelessness
    - Sabotage
  - We must ensure that the database can be recovered to a consistent state to prevent inconsistencies and data loss.

8.2 Transaction Support

- Transaction Support
  - Transactions
    - We view a database transaction as a *logical unit of work.*
      - The unit of work may involve any number of database operations
        - E.g., a unit of work may be comprised of a SELECT operation followed by an UPDATE operation followed by an INSERT operation followed by another SELECT operation followed by a DELETE operation. The operations need **not** be applied to the same table.
    - Transactions should always transform the database from one consistent state to another consistent state.
      - Whilst the transaction is in progress the database may effectively be in an inconsistent state.

Can you think of an example that describes how this can occur?

8.2 Transaction Support

- Transaction Support
  - Transaction Outcomes

Success

Failure

If the transaction is successful then we say that the transaction has been **committed** to the database.

If the transaction does not execute successful then the transaction is **aborted.** In this case, the database must be restored to the consistent state it was in prior to the transaction commencing. We say that the transaction is **rolled-back** or **undone** in this case.

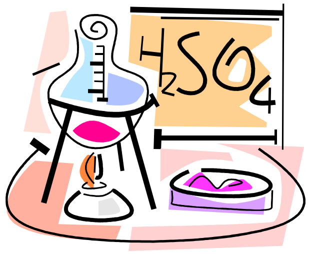
Note: A committed transaction can not be rolled-back. If an error has been made and committed to the database then it may be necessary to perform a **compensating transaction.**

8.2 Transaction Support

- Transaction Support
  - Specifying a Logical Unit of Work
    - The DBMS requires some indication as to what group of database operations should be considered a logical unit of work
    - We must supply an indication of where the transaction boundaries lie
    - Typically, a DBMS will provide a set of methods to delimit transaction boundaries:
      - BEGIN TRANSACTION
      - COMMIT
      - ROLLBACK

8.2 Transaction Support

- Transaction Support
  - Transaction Properties - **ACID**



Property	Description
ATOMICITY	All or Nothing – either a transaction is performed in its entirety or it is not performed at all.
CONSISTENCY	A transaction must transform the database from one consistent state to another consistent state.
ISOLATION	Transactions execute independently of each other. The partial effects of incomplete transactions should not be visible to other transactions.
DURABILITY	The effects of a committed transaction are permanently recorded in the database and must not be lost because of any subsequent failure.

8.3 Concurrency Control

- Concurrency Control
  - A database should be capable of being used as a shared data resource between many users and different client application programs
  - If each of the client applications or users are accessing the database solely to *read* data then there will be no way that the clients can interfere with each other.
  - If, however, two or more processes are accessing the database and at least one of the processes is updating (inserting, deleting) data, then there may be the chance that the processes interfere with each other resulting in data inconsistencies.

8.3 Concurrency Control

- Concurrency Control
  - The situation is similar to the way modern operating systems have the capability to implement a *multiprogramming* environment
    - A multiprogramming environment allows the operating system to carry out operations associated with different processes simultaneously
    - This occurs through the process of *interleaving* the processes to make efficient use of the CPU
    - In this case the *throughput* that can be achieved is greatly increased
  - For a database system we can encounter the situation where two separate transactions may be perfectly fine in and of themselves, however, if the transaction operations are interleaved we may produce undesirable results that affect the integrity and consistency of the database.
  - We will look at the following examples of potential problems that can be caused by concurrent access to a database system:
    - The **lost update problem**
    - The **uncommitted dependency problem**
    - The **inconsistent analysis problem**

8.3 Concurrency Control

- Concurrency Control
  - We will look at the following examples of potential problems that can be caused by concurrent access to a database system:
    - The **Lost Update Problem**
    - The **Uncommitted Dependency Problem (aka Dirty Read)**
    - The **Inconsistent Analysis Problem (aka Non-Repeatable Read)**
    - The **Phantom Read Problem**
  - For the purpose of the examples that follow we will consider a simple bank account relation containing account balance information

8.3 Concurrency Control

- The Lost Update Problem
  - The Lost Update Problem occurs when two processes attempt to update the same data at the same time. This effectively creates a race-condition.
    - The process that updates the data last 'wins'
    - The process that updates the data first 'loses' and its data is overwritten by the second process

8.3 Concurrency Control

- The Lost Update Problem
  - Consider the following two transactions
    - T1: this transaction will attempt to withdraw €10 from an account whose balance is initially €100
    - T2: this transaction will attempt to deposit €100 into the same account
  - If transactions T1 and T2 are executed in a serial fashion (i.e., one after the other with no interleaving) then the final balance on the account will be €190 no matter which of the transactions is executed first.

8.3 Concurrency Control

- The Lost Update Problem
  - If transactions T1 and T2 are submitted to the database at the same time, however, we may encounter the lost update phenomenon.

Time	Transaction T1	Transaction T2	Balance	Narrative
t1	BEGIN TRANS		€100	The initial account balance is €100
t2	Read A/c 1 Balance	BEGIN TRANS	€100	T1 reads the account balance from the database as €100
t3	Add 100 to Balance	Read A/c 1 Balance	€100	T2 reads the account balance from the database as €100
t4	Write A/c 1 Balance	Subtract 10 from Balance	€200	T1 updates the account balance to €200. T2 subtracts €10 from the account balance previously read from the database as €100
t5	COMMIT TRANS	Write A/c 1 Balance	€90	T2 updates the account balance to €90
t6		COMMIT TRANS	€90	

8.3 Concurrency Control

- The Uncommitted Dependency Problem
  - The Uncommitted Dependency Problem occurs when an application process tries to read uncommitted data.
    - In this case an application process is allowed to see the intermediate results of another transaction prior to that other transaction being committed to the database
  - Consider the following two transactions
    - T3: this transaction will attempt to withdraw €10 from an account
    - T4: this transaction will attempt to deposit €100 into the same account but the transaction is aborted at a subsequent stage

8.3 Concurrency Control

- The Uncommitted Dependency Problem
  - We assume again that the initial account balance is €100 and that T4 begins execution prior to T3

Time	Transaction T3	Transaction T4	Balance	Narrative
t1		BEGIN TRANS	€100	The initial account balance is €100
t2		Read A/c 1 Balance	€100	T4 reads the account balance from the database as €100
t3		Add 100 to Balance	€100	T4 adds €100 to the balance
t4	BEGIN TRANS	Write A/c 1 Balance	€200	T4 updates the account balance to €200.
t5	Read A/c 1 Balance		€200	T3 reads the account balance from the database as €200
t6	Subtract 10 from Balance	ABORT TRANS	€200	T3 subtracts €10 from the balance. T4 aborts at this stage.
t7	Write A/c 1 Balance			T3 updates the account balance to €190
t8	COMMIT TRANS			

Here T3 is assuming that the T4 transaction will complete successfully

8.3 Concurrency Control

- The Inconsistent Analysis Problem

  - The Inconsistent Analysis Problem occurs when an application process submits the same query in a transaction but gets different results each time.
    - This shows that transactions that solely read the database can also produce inaccurate results if they are allowed read the partial results of other incomplete transactions that are engaged in updating the database

  - Consider the following two transactions

    - T5: this transaction will attempt to withdraw €10 from an account X and then lodge €10 to an account Z

    - T6: this transaction will attempt to total the balances of the three accounts X, Y, and Z

8.3 Concurrency Control

- The Inconsistent Analysis Problem

Time	Transaction T5	Transaction T6	A/c X Bal	A/c Y Bal	A/c Z Bal	Sum of A/c X, Y, & Z Balances
t1		BEGIN TRANS	€100	€50	€25	
t2	BEGIN TRANS	Sum = 0	€100	€50	€25	0
t3	Read A/c X Balance	Read A/c X Balance	€100	€50	€25	0
t4	Subtract €10 from A/c X Balance	Sum = Sum + A/c X Balance	€100	€50	€25	€100
t5	Write A/c X Balance	Read A/c Y Balance	€90	€50	€25	€100
t6	Read A/c Z Balance	Sum = Sum + A/c Y Balance	€90	€50	€25	€150
t7	Add €10 to A/c Z Balance		€90	€50	€25	€150
t8	Write A/c Z Balance		€90	€50	€35	€150
t9	COMMIT	Read A/c Z Balance	€90	€50	€35	€150
t10		Sum = Sum + A/c Z Balance	€90	€50	€35	€185
t11		COMMIT	€90	€50	€35	€185

8.3 Concurrency Control

- The Phantom Read Problem
  - The Phantom Read Problem occurs when an application process submits the same query in a transaction but gets additional rows in the result set each time.
    - Subsequent fetches of the same data in the same transaction return additional rows that were not in the original result set.

8.3 Concurrency Control

- Serializability and Recoverability
  - We would like to be able to schedule transactions in such a way that we do not encounter interference problems.
  - One solution to the problem is to implement a concurrency control protocol that ensures that only one transaction can execute against the database at any particular time
    - In this case one transaction must be committed prior to another transaction being permitted to begin
    - The drawback of this approach is that we are not maximising the degree of concurrency and parallelism associated with the DBMS.
      - For example, if two transactions are querying/updating completely separate tables in the database there would be no chance of interference yet one of the transactions will still have to wait for the other to complete before it can execute.

8.3 Concurrency Control

- Serializability and Recoverability

Schedule: A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions

- A transaction consists of a sequence of operations to the database followed by a commit or abort action.
- A *schedule* S consists of a sequence of the operations from a set of *n* transactions T1, T2, T3, …, Tn, subject to the constraint that the order of the operations is preserved in the schedule.

8.3 Concurrency Control

- Serializability and Recoverability

Serial Schedule: A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.

- The transactions are performed in serial order.
  - E.g., For two transactions T1 and T2 the order will either be T1 first followed by T2 OR T2 first followed by T1.
- One of the transactions must complete prior to the other commencing ensuring that there is no interference between the transactions

- Note: this does not necessarily mean that the results will be the same regardless of which order the transactions are executed in.

8.3 Concurrency Control

- Serializability and Recoverability

Non-Serial Schedule: A schedule where the operations of a set of concurrent transactions are interleaved.

- ☐ The purpose of *serializability* is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another.
- ☐ In this case the database will be left in a state that is equivalent to the state the database would be left in if the transactions had actually executed serially.
- ☐ If a set of transactions executes concurrently, we say that the non-serial schedule is ***correct*** *if it produces the same results as some serial execution*. In this case we say that the schedule is ***serializable.***

8.3 Concurrency Control

- Serializability and Recoverability
  - For serializability the order of read and write operations is important
    - If two transactions are solely engaged in reading data then there will be no conflict and the order does not matter
    - If two transactions are engaged in either reading or writing separate data items they will not conflict and again the order is not important
    - If a transactions writes a data item and another transaction reads from or writes to that same data item then the order of execution is important.

In this case the operations are said to conflict

8.3 Concurrency Control

- Serializability and Recoverability

This type of serializability is called ***conflict serializability.*** A conflict serializable schedule orders any conflicting operations in the same way as some serial execution.

Schedule S1

T1	T2
BEGIN TRANS	
Read Bal A/c X	
Write Bal A/c X	
	BEGIN TRANS
	Read Bal A/c X
	Write Bal A/c X
Read Bal A/c Y	
Write Bal A/c Y	
COMMIT	
	Read Bal A/c Y
	Write Bal A/c Y
	COMMIT

Schedule S2

T1	T2
BEGIN TRANS	
Read Bal A/c X	
Write Bal A/c X	
	BEGIN TRANS
	Read Bal A/c X
Read Bal A/c Y	
	Write Bal A/c X
Write Bal A/c Y	
COMMIT	
	Read Bal A/c Y
	Write Bal A/c Y
	COMMIT

Schedule S3

T1	T2
BEGIN TRANS	
Read Bal A/c X	
Write Bal A/c X	
Read Bal A/c Y	
Write Bal A/c Y	
COMMIT	
	BEGIN TRANS
	Read Bal A/c X
	Write Bal A/c X
	Read Bal A/c Y
	Write Bal A/c Y
	COMMIT

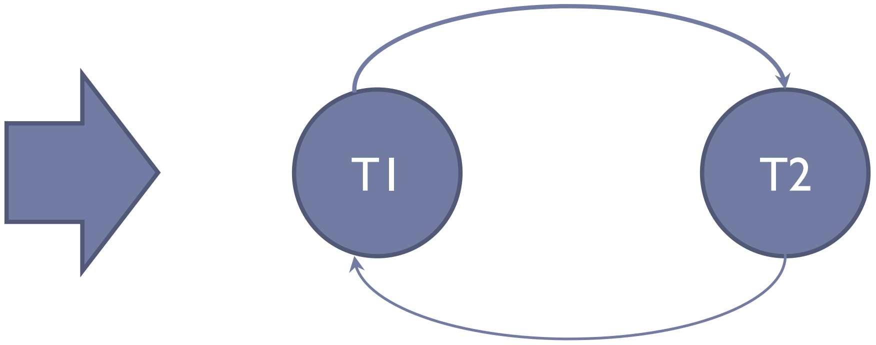
8.3 Concurrency Control

- Serializability and Recoverability
  - Conflict Graph / Precedence Graph
    - Graph consists of
      - A node for each transaction
      - A directed edge T*i* → T*j* if T*j* reads the value of an item written by T*i*
      - A directed edge T*i* → T*j* if T*j* writes the value into an item after it has been read by T*i*
    - If the graph contains a cycle then the schedule is **not** conflict serializable

8.3 Concurrency Control

- Serializability and Recoverability
  - Conflict Graph / Precedence Graph Example
    - Consider the transactions T1 and T2

T1	T2
BEGIN TRANS	
Read Bal A/c X	
Add €100 to A/c X Bal	
Write A/c X Bal	BEGIN TRANS
	Read Bal A/c X
	Add €10 to A/c X Bal
	Write A/c X Bal
	Read Bal A/c Y
	Add €10 to A/c Y Bal
	Write A/c Y Bal
Read Bal A/c Y	COMMIT
Subtract €100 from A/c Y Bal	
Write A/c Y Bal	
COMMIT	


$$\begin{tikzcd} T1 \arrow[r, bend left] & T2 \arrow[l, bend left] \end{tikzcd}$$

There is a cycle in the graph => this schedule is not conflict serializable

8.3 Concurrency Control

- Serializability and Recoverability
  - Note: there are other types of serializability that do not require such stringent definitions of schedule equivalence to that used by conflict serializability – e.g ***View Serializability***
  - View Serializability
    - Two schedules S1 and S2 consisting of the same operations from a set of *n* transactions T1, T2, …, T*n* are said to be *view equivalent* if the following three conditions hold:
      - For each data item *x*, if transaction T*i* reads the initial value of *x* in schedule S1, then transaction T*i* must also read the initial value of *x* in schedule S2
      - For each read operation on data item *x* by transaction T*i* in schedule S1, if the value read by *x* has been written by transaction T*j*, then transaction T*i* must also read the value of *x* produced by transaction T*j* in schedule S2
      - For each data item *x*, if the last write operation on *x* was performed by transaction T*i* in schedule S1, the same transaction must perform the final write on data item *x* in schedule S2.
    - A schedule is *view serializable* if it is equivalent to a serial schedule.
  - In practice, a DBMS does not test for the serializability of a schedule. Instead, certain protocols can be implemented that will produce serializable schedules.

8.3 Concurrency Control

- Serializability and Recoverability

  - Recoverability
    - We can also take another perspective when examining schedules to ascertain whether or not the consistency of the database will be maintained.
    - We look at the *recoverability* of transactions within a schedule.

    - Recall the ACID properties
      - A -> Atomicity -> All or Nothing
      - D -> Durability -> once transaction has committed it can not be undone

    - When looking at the recoverability of transactions within a schedule we examine the consequences of one of the transactions aborting with a rollback of the effects of the transaction.

8.3 Concurrency Control

- Serializability and Recoverability

  - Recoverability

T1	T2
BEGIN TRANS	
Read Bal A/c X	
Add €100 to A/c X Bal	
Write A/c X Bal	BEGIN TRANS
	Read Bal A/c X
	Add €10 to A/c X Bal
	Write A/c X Bal
	Read Bal A/c Y
	Add €10 to A/c Y Bal
	Write A/c Y Bal
Read Bal A/c Y	COMMIT
Subtract €100 from A/c Y Bal	
Write A/c Y Bal	
ROLLBACK	

In this case (due to the durability of transactions) A/c X's balance value will be incorrect.

This is an example of a non-recoverable schedule

Recoverable Schedule: A schedule where, for each pair of transactions T*i* and T*j*, if T*j* reads a data item previously written by T*i*, then the COMMIT operation of T*i* precedes the COMMIT operation of T*j*.

8.4 Summary

- Introduction
- Transaction Support
- Concurrency Control