# DATABASE MANAGEMENT SYSTEM

# Chapter 10
# Data Evaluation and Transaction

**Bikash Khadka Shah**

MCSE, MSDA, OCP, RHCE, RHCVA, CCNA, CEH

9841766620 | 9801076620

# Transactions

## Definition

- A transaction is a collection of one or more operations on one or more databases, which reflects a single real-world transition
  - In the real world, this happened (completely) or it didn't happen at all (Atomicity)
- Commerce examples
  - Transfer money between accounts
  - Purchase a group of products
- Student record system
  - Register for a class (either waitlist or allocated)

## Coding a transaction

- Typically a computer-based system doing OLTP has a collection of *application programs*
- Each program is written in a high-level language, which calls DBMS to perform individual SQL statements
  - Either through embedded SQL converted by preprocessor
  - Or through Call Level Interface where application constructs appropriate string and passes it to DBMS

## Why write programs?

- Why not just write a SQL statement to express “what you want”?
- An individual SQL statement can’t do enough
  - It can’t update multiple tables
  - It can’t perform complicated logic (conditionals, looping, etc)

## COMMIT

- As app program is executing, it is “in a transaction”
- Program can execute COMMIT
  - SQL command to finish the transaction successfully
  - The next SQL statement will automatically start a new transaction

## Warning

- The idea of a transaction is hard to see when interacting directly with DBMS, instead of from an app program
- Using an interactive query interface to DBMS, by default each SQL statement is treated as a separate transaction (with implicit COMMIT at end) unless you explicitly say “START TRANSACTION”

## A Limitation

- Some systems rule out having both DML and DDL statements in a single transaction
- i.e., you can change the schema, or change the data, but not both

## ROLLBACK

- If the app gets to a place where it can’t complete the transaction successfully, it can execute ROLLBACK
- This causes the system to “abort” the transaction
  - The database returns to the state without any of the previous changes made by activity of the transaction

## Reasons for Rollback

- User changes their mind (“ctl-C”/cancel)
- Explicit in program, when app program finds a problem
  - e.g. when qty on hand < qty being sold
- System-initiated abort
  - System crash
  - Housekeeping
    - e.g. due to timeouts

## Atomicity

- Two possible outcomes for a transaction
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made
- That is, transaction's activities are all or nothing

## Integrity

- A real world state is reflected by collections of values in the tables of the DBMS
- But not every collection of values in a table makes sense in the real world
- The state of the tables is restricted by integrity constraints
- e.g. account number is unique
- e.g. stock amount can’t be negative

## Integrity

- Many constraints are explicitly declared in the schema
  - So the DBMS will enforce them
  - Especially: primary key (some column's values are non null, and different in every row)
  - And referential integrity: value of foreign key column is actually found in another "referenced" table
- Some constraints are not declared
  - They are business rules that are supposed to hold

## Consistency

- Each transaction can be written on the assumption that all integrity constraints hold in the data, before the transaction runs
- It must make sure that its changes leave the integrity constraints still holding
  - However, there are allowed to be intermediate states where the constraints do not hold
- A transaction that does this, is called consistent
- This is an obligation on the programmer
  - Usually the organization has a testing/checking and sign-off mechanism before an application program is allowed to get installed in the production system

## System obligations

- Provided the app programs have been written properly,
- Then the DBMS is supposed to make sure that the state of the data in the DBMS reflects the real world accurately, as affected by all the committed transactions

## Local to global reasoning

- Organization checks each app program as a separate task
  - Each app program running on its own moves from state where integrity constraints are valid to another state where they are valid
- System makes sure there are no nasty interactions
- So the final state of the data will satisfy all the integrity constraints

## Example - Tables

- System for managing inventory
- InStore(prodID, storeID, qty)
- Product(prodID, desc, mnfr, …, WarehouseQty)
- Order(orderNo, prodID, qty, rcvd, ….)
  - Rows never deleted!
  - Until goods received, rcvd is null
- Also Store, Staff, etc etc

## Example - Constraints

- Primary keys
  - InStore: (prodID, storeID)
  - Product: prodID
  - Order: orderId
  - etc
- Foreign keys
  - Instore.prodID references Product.prodID
  - etc

## Example - Constraints

- Data values
  - Instore.qty >= 0
  - Order.rcvd <= current_date or Order.rcvd is null
- Business rules
  - for each p, (Sum of qty for product p among all stores and warehouse) >= 50
  - for each p, (Sum of qty for product p among all stores and warehouse) >= 70 or there is an outstanding order of product p

## Example - transactions

- MakeSale(store, product, qty)
- AcceptReturn(store, product, qty)
- RcvOrder(order)
- Restock(store, product, qty)
  - // move from warehouse to store
- ClearOut(store, product)
  - // move all held from store to warehouse
- Transfer(from, to, product, qty)
  - // move goods between stores

## Example - ClearOut

- Validate Input (appropriate product, store)
- SELECT qty INTO :tmp
  FROM InStore
  WHERE StoreID = :store AND prodID = :product
- UPDATE Product
  SET WarehouseQty = WarehouseQty + :tmp
  WHERE prodID = :product
- UPDATE InStore
  SET Qty = 0
  WHERE prodID = :product
- COMMIT

This is one way to write
the application; other algorithms
are also possible

## Example - Restock

- Input validation
  - Valid product, store, qty
  - Amount of product in warehouse >= qty
- UPDATE Product

  SET WarehouseQty = WarehouseQty - :qty

  WHERE prodID = :product
- If no record yet for product in store

  INSERT INTO InStore (:product, :store, :qty)
- Else, UPDATE InStore

      SET qty = qty + :qty

      WHERE prodID = :product and storeID = :store
- COMMIT

# Example - Consistency

- How to write the app to keep integrity holding?
- MakeSale logic:
  - Reduce Instore.qty
  - Calculate sum over all stores and warehouse
  - If sum < 50, then ROLLBACK // Sale fails
  - If sum < 70, check for order where date is null
    - If none found, insert new order for say 25
  - COMMIT

This terminates execution of the program (like return)

## Example - Consistency

- We don’t need any fancy logic for checking the business rules in Restock, ClearOut, Transfer
  - Because sum of qty not changed; presence of order not changed
    - *provided integrity holds before txn, it will still hold afterwards*
- We don’t need fancy logic to check business rules in AcceptReturn
  - *why?*
- Is checking logic needed for RcvOrder?

## Threats to data integrity

- Need for application rollback
- System crash
- Concurrent activity

- The system has mechanisms to handle these

## Application rollback

- A transaction may have made changes to the data before discovering that these aren’t appropriate
  - the data is in state where integrity constraints are false
  - Application executes ROLLBACK
- System must somehow return to earlier state
  - Where integrity constraints hold
- So aborted transaction has no effect at all

## Example

- While running MakeSale, app changes InStore to reduce qty, then checks new sum
- If the new sum is below 50, txn aborts
- System must change InStore to restore previous value of qty
  - Somewhere, system must remember what the previous value was!

## System crash

- At time of crash, an application program may be part-way through (and the data may not meet integrity constraints)
- Also, buffering can cause problems
  - Note that system crash loses all buffered data, restart has only disk state
  - Effects of a committed txn may be only in buffer, not yet recorded in disk state
  - Lack of coordination between flushes of different buffered pages, so even if current state satisfies constraints, the disk state may not

## Example

- Suppose crash occurs after
  - MakeSale has reduced InStore.qty
  - found that new sum is 65
  - found there is no unfilled order
  - // but before it has inserted new order
- At time of crash, integrity constraint did not hold
- Restart process must clean this up (effectively aborting the txn that was in progress when the crash happened)

## Concurrency

- When operations of concurrent threads are interleaved, the effect on shared state can be unexpected
- Well known issue in operating systems, thread programming
  - see OS textbooks on critical section
  - Java use of synchronized keyword

## Famous anomalies

- Dirty data
  - One task T reads data written by T’ while T’ is running, then T’ aborts (so its data was not appropriate)
- Lost update
  - Two tasks T and T’ both modify the same data
  - T and T’ both commit
  - Final state shows effects of only T, but not of T’
- Inconsistent read
  - One task T sees some but not all changes made by T’
  - The values observed may not satisfy integrity constraints
  - This was not considered by the programmer, so code moves into absurd path

## Example – Dirty data

AcceptReturn(p1,s1,50)
MakeSale(p1,s2,65)

Update row 1: 25 -> 75

update row 2: 70->5

find sum: 90

// no need to insert

// row in Order

Abort

// rollback row 1 to 25

COMMIT

Integrity constraint is false:
Sum for p1 is only 40!

| | | |
|---|---|---|
| p1 | s1 | 25 |
| p1 | s2 | 70 |
| p2 | s1 | 60 |
| etc | etc | etc |

| | | |
|---|---|---|
| p1 | etc | 10 |
| p2 | etc | 44 |
| etc | etc | etc |

Initial state of InStore, Product

| | | |
|---|---|---|
| p1 | s1 | 25 |
| p1 | s2 | 5 |
| p2 | s1 | 60 |
| etc | etc | etc |

| | | |
|---|---|---|
| p1 | etc | 10 |
| p2 | etc | 44 |
| etc | etc | etc |

Final state of InStore, Product

# Example – Lost update

- ClearOut(p1,s1)
  AcceptReturn(p1,s1,60)
- Query InStore; qty is 25
- Add 25 to WarehouseQty: 40->65
- Update row 1: 25->85
- Update row 1, setting it to 0
- COMMIT
- COMMIT

60 returned p1's have vanished from system; total is still 135

| | | |
|---|---|---|
| p1 | s1 | 25 |
| p1 | s2 | 50 |
| p2 | s1 | 45 |
| etc | etc | etc |

| | | |
|---|---|---|
| p1 | etc | 40 |
| p2 | etc | 55 |
| etc | etc | etc |

Initial state of InStore, Product

| | | |
|---|---|---|
| p1 | s1 | 0 |
| p1 | s2 | 50 |
| p2 | s1 | 45 |
| etc | etc | etc |

| | | |
|---|---|---|
| p1 | etc | 65 |
| p2 | etc | 55 |
| etc | etc | etc |

Final state of InStore, Product

# Example – Inconsistent read

- ClearOut(p1,s1) MakeSale(p1,s2,60)
- Query InStore: qty is 30
- Add 30 to WarehouseQty: 10->40
- update row 2: 65->5
- find sum: 75
- // no need to insert
- // row in Order
- Update row 1, setting it to 0
- COMMIT
- COMMIT

Integrity constraint is false:
Sum for p1 is only 45!

| | | |
|---|---|---|
| p1 | s1 | 30 |
| p1 | s2 | 65 |
| p2 | s1 | 60 |
| etc | etc | etc |

| | | |
|---|---|---|
| p1 | etc | 10 |
| p2 | etc | 44 |
| etc | etc | etc |

Initial state of InStore, Product

| | | |
|---|---|---|
| p1 | s1 | 0 |
| p1 | s2 | 5 |
| p2 | s1 | 60 |
| etc | etc | etc |

| | | |
|---|---|---|
| p1 | etc | 40 |
| p2 | etc | 44 |
| etc | etc | etc |

Final state of InStore, Product

## Serializability

- To make isolation precise, we say that an execution is serializable when
- There exists some serial (ie batch, no overlap at all) execution of the same transactions which has the same final state
  - Hopefully, the real execution runs faster than the serial one!
- NB: different serial txn orders may behave differently; we ask that *some* serial order produces the given state
  - Other serial orders may give different final states

# Example – Serializable execution

- ClearOut(p1,s1) MakeSale(p1,s2,20)
- Query InStore: qty is 30
- update row 2: 45->25
- find sum: 65
- no order for p1 yet
- Add 30 to WarehouseQty: 10->40
- Update row 1, setting it to 0
- COMMIT
- Insert order for p1

COMMIT

Execution is like serial MakeSale; ClearOut

| | | |
|---|---|---|
| p1 | s1 | 30 |
| p1 | s2 | 45 |
| p2 | s1 | 60 |
| etc | etc | etc |

| | | |
|---|---|---|
| p1 | etc | 10 |
| p2 | etc | 44 |
| etc | etc | etc |

Order: empty

Initial state of InStore, Product, Order

| | | |
|---|---|---|
| p1 | s1 | 0 |
| p1 | s2 | 25 |
| p2 | s1 | 60 |
| etc | etc | etc |

| | | |
|---|---|---|
| p1 | etc | 40 |
| p2 | etc | 44 |
| etc | etc | etc |

| | | | |
|---|---|---|---|
| p1 | 25 | Null | etc |

Final state of InStore, Product, Order

## Serializability Theory

- There is a beautiful mathematical theory, based on formal languages
  - Treat the set of all serializable executions as an object of interest (called SR)
  - Thm: SR is in NP, i.e. the task of testing whether an execution is serializable seems unreasonably slow
- Does it matter?
  - The goal of practical importance is to design a system that produces some subset of the collection of serializable executions
  - It's not clear that we care about testing arbitrary executions that don't arise in our system

## Conflict serializability

- There is a nice sufficient condition (ie a conservative approximation) called conflict serializable, which can be efficiently tested
  - Draw a precedes graph whose nodes are the transactions
  - Edge from Ti to Tj when Ti accesses x, then later Tj accesses x, and the accesses conflict (not both reads)
  - The execution is conflict serializable iff the graph is acyclic
- Thm: if an execution is conflict serializable then it is serializable
  - Pf: the serial order with same final state is any topological sort of the precedes graph
- Most people and books use the approximation, usually without mentioning it!

## ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

## Transaction: Atomicity

- The atomicity property of a transaction requires that all operations of a transaction be completed, if not, the transaction is aborted.
- In other words, a transaction is treated as single, individual logical unit of work.
- Therefore, a transaction must execute and complete each operation in its logic before it commits its changes.
- As the transaction is considered as one operation even though there are multiple read and writes it completes or fails as one unit.
- The atomicity property of transaction is ensured by the transaction recovery subsystem of a DBMS.
- In the event of a system crash in the midst of transaction execution, the recovery techniques undo any effects of the transaction on the database.

## Transaction: Consistency

- Execution of a transaction must leave a database in either its prior stable state or a new stable state that reflects the new modifications (updates) made by the transaction.
  - If the transaction fails, the database must be returned to the state it was in prior to the execution of the failed transaction.
  - If the transaction commits, the database must reflect the new changes.
- Thus, all resources are always in a consistent state.
- In other words, a transaction must transform the database from one consistent state to another consistent state.
- The preservation of consistency is generally the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints.

## Transaction: Isolation

- Isolation property of a transaction means that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.
- This property isolates transactions from one another.
- In other words, if a transaction T1 is being executed and is using the data item X, that data item cannot be accessed by any other transaction (T2……….Tn) until T1 ends.
- The isolation property is enforced by the concurrency control subsystem of the DBMS.

## Transaction: Durability

- The durability property of transaction indicates the performance of the database's consistent state.
- It states that the changes made by a transaction are permanent.
- They cannot be lost by either a system failure or by the erroneous operation of a faulty transaction.
- When a transaction is completed, the database reaches a consistent state and that state cannot be lost, even in the event of system's failure.
- Durability property is the responsibility of the recovery subsystem of the DBMS.

## Single User vs Multiuser System

- A DBMS is single-user if at most one user at a time can use the system.
- It is multi user if many users can use the system at a time.
  - So they can access the database concurrently.
- Single user DBMSs are mostly restricted to a PC; most of the DBMSs are multiuser and many of the users would be accessing them for business process continuation.

## Concurrency control

- Concurrency control and recovery mechanisms are mainly concerned with the database commands in the transaction.
- Transactions submitted by the various users may execute concurrently and may access and update the same database items.
- If the concurrent execution is uncontrolled, it may lead to problems:
  - The lost update problem
  - The temporary update (or dirty read) problem
  - The incorrect summary problem
  - The unrepeatable read problem

## Concurrency control...

- Lost update problem occurs when two transactions that access the same database items have interleaved in a way that makes the value of some db item incorrect.
- Dirty read problem occurs when one transaction updates a database item and fails for some reason. Meanwhile, the updated db item is read by another transaction before it is changed back to its original value.
- If one transaction is calculating an aggregate summary function on a number of db items while other transactions are updating some of these items, the incorrect summary problem occurs.
- In unrepeatable read, a transaction T reads the same item twice and the item is changed by another transaction T1 between the two reads.

## Concurrency Control Methods

- Concurrency control is provided in a database to:
  - enforce isolation among transactions.
  - preserve database consistency through consistency preserving execution of transactions.
  - resolve read-write and write-read conflicts.
- Concurrency control protocols can be broadly divided into two categories −
  - Lock based protocols
  - Time stamp based protocols

## Lock Based Protocols

- A lock is a variable associated with a data item that describes a status of data item with respect to possible operation that can be applied to it.
- They synchronize the access by concurrent transactions to the database items.
- In lock based protocols, It is required that all the data items must be accessed in a mutually exclusive manner.
- Shared Lock (S):
  - Also known as Read-only lock.
  - As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item.
- Exclusive Lock (X):
  - Data item can be both read as well as written.
  - This is Exclusive and cannot be held simultaneously on the same data item.
  - X-lock is requested using lock-X instruction.

## Lock compatibility matrix:

| | S | X |
|---|---|---|
| S | ☑ | ☒ |
| X | ☒ | ☒ |

- **Lock conversion:**
  - A transaction that holds a lock on an item A is allowed under certain condition to change the lock state from one state to another.
  - Upgrade: A S(A) can be upgraded to X(A) if Ti is the only transaction holding the S-lock on element A.
  - Downgrade: We may downgrade X(A) to S(A) when we feel that we no longer want to write on data-item A. As we were holding X-lock on A, we need not check any conditions.

## Lock Based Protocols…

- Simple locking may not always produce Serializable results, it may lead to Deadlock.
  - –It allows transactions to obtain a lock on every object before beginning operation.
  - –Transactions may unlock the data item after finishing the 'write' operation.
- However these may cause some problems, eg:

| | T1 | T2 |
|---|---|---|
| 1 | lock-X(B) | |
| 2 | read(B) | |
| 3 | B:=B-50 | |
| 4 | write(B) | |
| 5 | | lock-S(A) |
| 6 | | read(A) |
| 7 | | lock-S(B) |
| 8 | lock-X(A) | |
| 9 | …… | …… |

## Lock Based Protocols…

- Deadlock:
  - Consider the above execution phase.
  - Now, T1 holds an Exclusive lock over B, and T2 holds a Shared lock over A.
  - Consider Statement 7, T2 requests for lock on B, while in Statement 8 T1 requests lock on A.
  - This as you may notice imposes a Deadlock as none can proceed with their execution.
- Starvation:
  - It is also possible if concurrency control manager is badly designed.
  - For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.

## Two Phase Locking

- A transaction is said to follow Two Phase Locking protocol if Locking and Unlocking can be done in two phases.
  - Growing Phase: New locks on data items may be acquired but none can be released.
  - Shrinking Phase: Existing locks may be released but no new locks can be acquired.
- If lock conversion is allowed, then upgrading of lock( from S(a) to X(a) ) is allowed in Growing Phase and downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.
- The Point at which the growing phase ends, i.e., when transaction takes the final lock it needs to carry on its work is known as Lock Point.
- 2PL ensures serializability, but cascading rollbacks and deadlocks are possible.

## Two Phase Locking…

| | T1 | T2 |
|---|---|---|
| 1 | LOCK-S(A) | |
| 2 | | LOCK-S(A) |
| 3 | LOCK-X(B) | |
| 4 | ……. | …… |
| 5 | UNLOCK(A) | |
| 6 | | LOCK-X(C) |
| 7 | UNLOCK(B) | |
| 8 | | UNLOCK(A) |
| 9 | | UNLOCK(C) |
| 10 | ……. | …… |

- A transaction implementing 2-PL.
- Transaction T1:
  - Growing Phase is from steps 1-3, Shrinking Phase is from steps 5-7.
  - Lock Point at 3
- Transaction T2:
  - Growing Phase is from steps 2-6, Shrinking Phase is from steps 8-9.
  - Lock Point at 6

## Two Phase Locking

- There are some variations of 2PL.
- Strict 2-PL requires that in addition to the lock being 2-Phase all Exclusive(X) Locks held by the transaction be released until after the Transaction Commits; ensuring cascade less and recoverable.
- Rigorous 2-PL requires that in addition to the lock being 2Phase all Exclusive(X) and Shared(S) Locks held by the transaction be released until after the Transaction Commits.
- Conservative 2-PL or Static 2-PL, requires the transaction to lock all the items it access before the Transaction begins execution by pre-declaring its read-set and write-set.
  - If any of the pre-declared items needed cannot be locked, the transaction does not lock any of the items, instead it waits until all the items are available for locking.

## Evaluation of DBMS

**To choose the most suitable DBMS, we need to evaluate from various systems. We perform a structured approach to evaluate the database systems.**

### Evaluation Methodology

### 1. Feature analysis

**In here, we determine whether the DBMS provides all the features required for the operations that are to be performed on the data and shortlist the DBMS. This analysis serves two purposes:**

- **It directly eliminates the DBMS that is not competent to fulfill all the needs of the user.**
- **We can rank the remaining systems based on the features provided by them.**

### 2. Performance Analysis

**In here, we analyze only those shortlisted DBMS, evaluate the systems' efficiency, and choose one with maximum efficiency. The purpose of this analysis is:**

- **To analyze the behavior of the database management system in the user system.**
- **To gather the data on the performance of the DBMS and rank them accordingly.**

## Database evaluation process

1. Planning the Database.
2. Analyzing Information for the Database.
3. Designing the Database.
4. Implementing the Database.
5. Testing the Database.
6. Maintaining the Database.

## Planning the Database

- Gather facts to understand the problem the database is trying to solve
- Specify the overall objectives of the database
- Deliverable: Database requirements statements

## Analyzing Information for the Database

- Identify functional and user requirements (what will be stored, how it will be used, expected reports)
- Deliverable: Database system specifications

## Designing the Database

- Design each technical component based on the system specifications
- Document in a technical specifications document
- Deliverable: Conceptual database technical specifications

## Implementing the Database

- Create the actual database design and install it
- May be phased to test each component
- Includes developing applications, testing, documentation, user training, populating data
- Deliverable: The operational database

## Testing the Database

- Test the database to ensure it meets all requirements
- Install on all required computers and test links between components
- Deliverable: Functional database ready for use

## Maintaining the Database

- Monitor database functioning to ensure continuous operation
- Proactive, preventive, corrective and adaptive maintenance tasks
- Allocate user access, routine monitoring, periodic testing
- Deliverable: Ongoing database with backed-up data that evolves to meet changing requirements