DSCE
Dept. of Information Science & Engg

Database Management System

(18IS5DCDBM)

Unit 5

(Database Design, Transaction Management)

Mrs. Bhavani K
Assistant Professor
Dept. of ISE, DSCE

DSCE
Dept. of Information Science & Engg

Unit 5

- **Database Design:**
  - General Definitions of Second and Third Normal Forms
  - Boyce-Codd Normal form

- **Transaction Management**:
  - The ACID properties
  - Transactions and Schedules
  - Concurrent Execution of Transactions
  - Lock Based Concurrency Control
  - Transaction Support in SQL

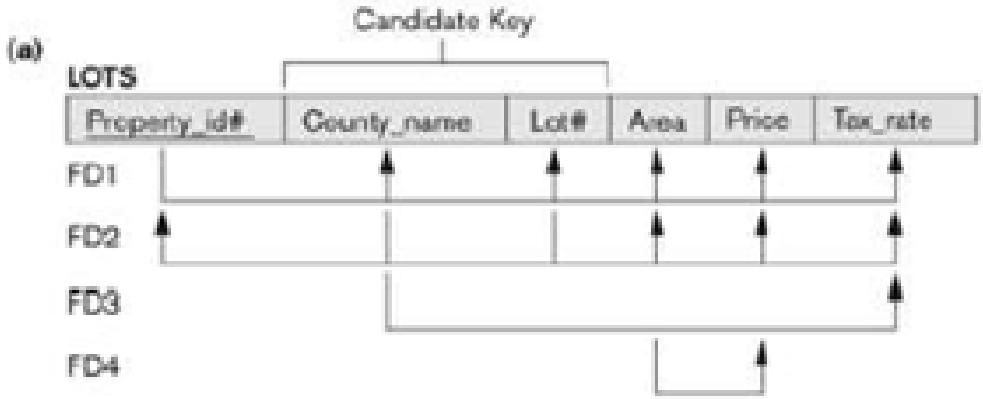
Database Design

Normal Forms Defined Informally

- 1st normal form
  - All attributes depend on the key
- 2nd normal form
  - All attributes depend on the whole key
- 3rd normal form
  - All attributes depend on nothing but the key

General Normal Form Definitions (For Multiple Keys)

- The above definitions consider the primary key only
- The following more general definitions take into account relations with multiple candidate keys
- Any attribute involved in a candidate key is a prime attribute
- All other attributes are called non-prime attributes.



Figure

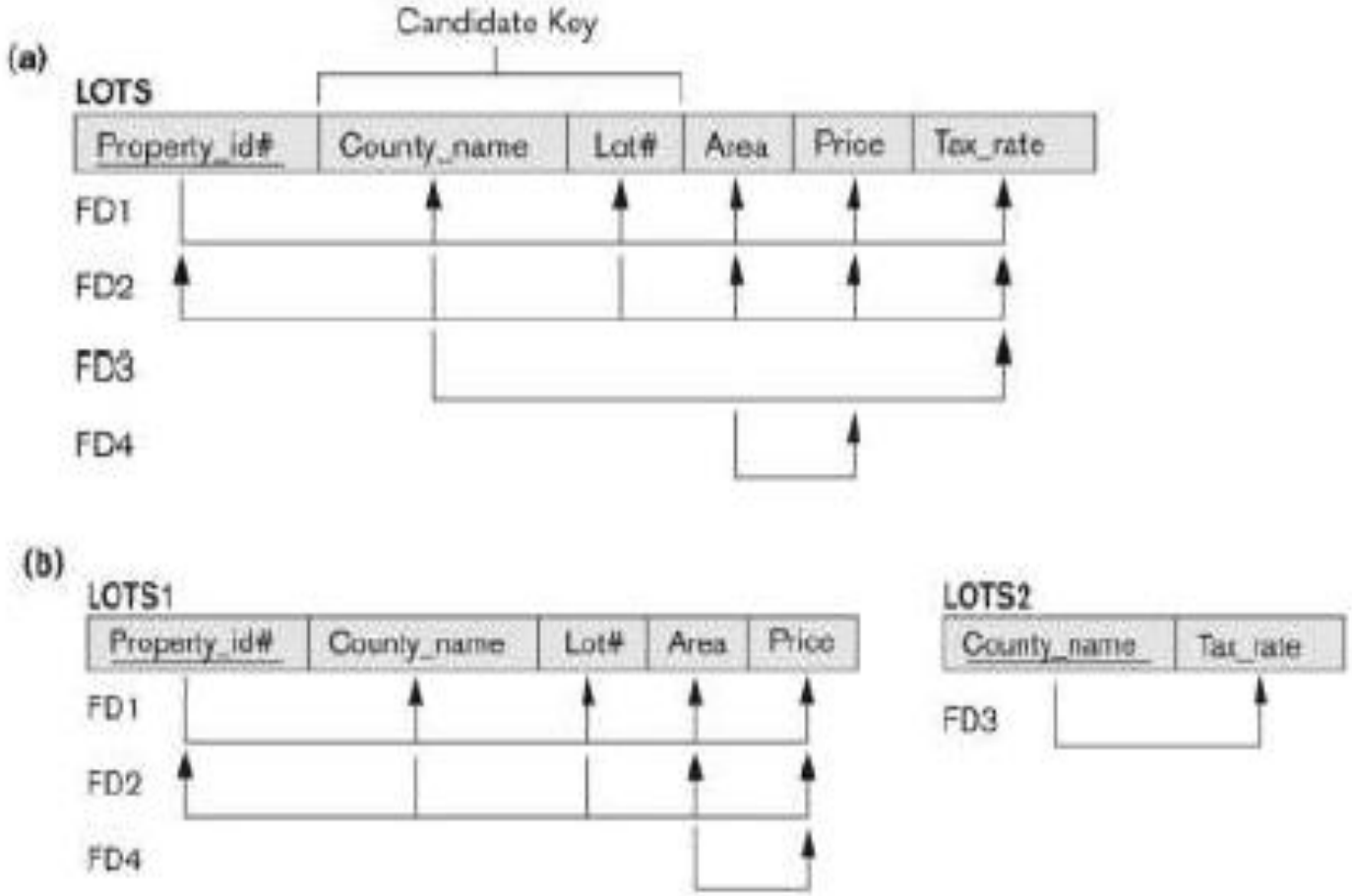
The LOTS relation with its functional dependencies FD1 through FD4.

General Definition of 2NF (For Multiple Candidate Keys)

- A relation schema R is in second normal form (2NF) if every non-prime attribute A in R is fully functionally dependent on every key of R

- In Figure , the FD
  - County_name → Tax_rate violates 2NF

- So second normalization converts LOTS into
  - LOTS1 (Property_id#, County_name, Lot#, Area, Price)
  - LOTS2 (County_name, Tax_rate)



Figure

Normalization into 2NF
(a) The LOTS relation with its functional dependencies FD1 through FD4.
(b) Decomposing into the 2NF relations LOTS1 and LOTS2.

General Definition of Third Normal Form

- Definition:
  - Superkey of relation schema R - a set of attributes S of R that contains a key of R
  - A relation schema R is in third normal form (3NF) if whenever a FD X → A holds in R, then either:
    - (a) X is a superkey of R, or
    - (b) A is a prime attribute of R
- LOTS1 relation violates 3NF because Area → Price ; and Area is not a superkey in LOTS1. (see Figure).

Interpreting the General Definition of Third Normal Form

- Consider the 2 conditions in the Definition of 3NF:
  - A relation schema R is in third normal form (3NF) if whenever a FD X → A holds in R, then either:
    - (a) X is a superkey of R, or
    - (b) A is a prime attribute of R

    - Condition (a) catches two types of violations :

      - one where a prime attribute functionally determines a non-prime attribute. This catches 2NF violations due to non-full functional dependencies.

      - second, where a non-prime attribute functionally determines a non-prime attribute. This catches 3NF violations due to a transitive dependency.

Interpreting the General Definition of Third Normal Form

- ALTERNATIVE DEFINITION of 3NF
  - We can restate the definition as:
    - A relation schema R is in third normal form (3NF) if every non-prime attribute in R meets both of these conditions:
      - It is fully functionally dependent on every key of R
      - It is non-transitively dependent on every key of R
    - Note that stated this way, a relation in 3NF also meets the requirements for 2NF.

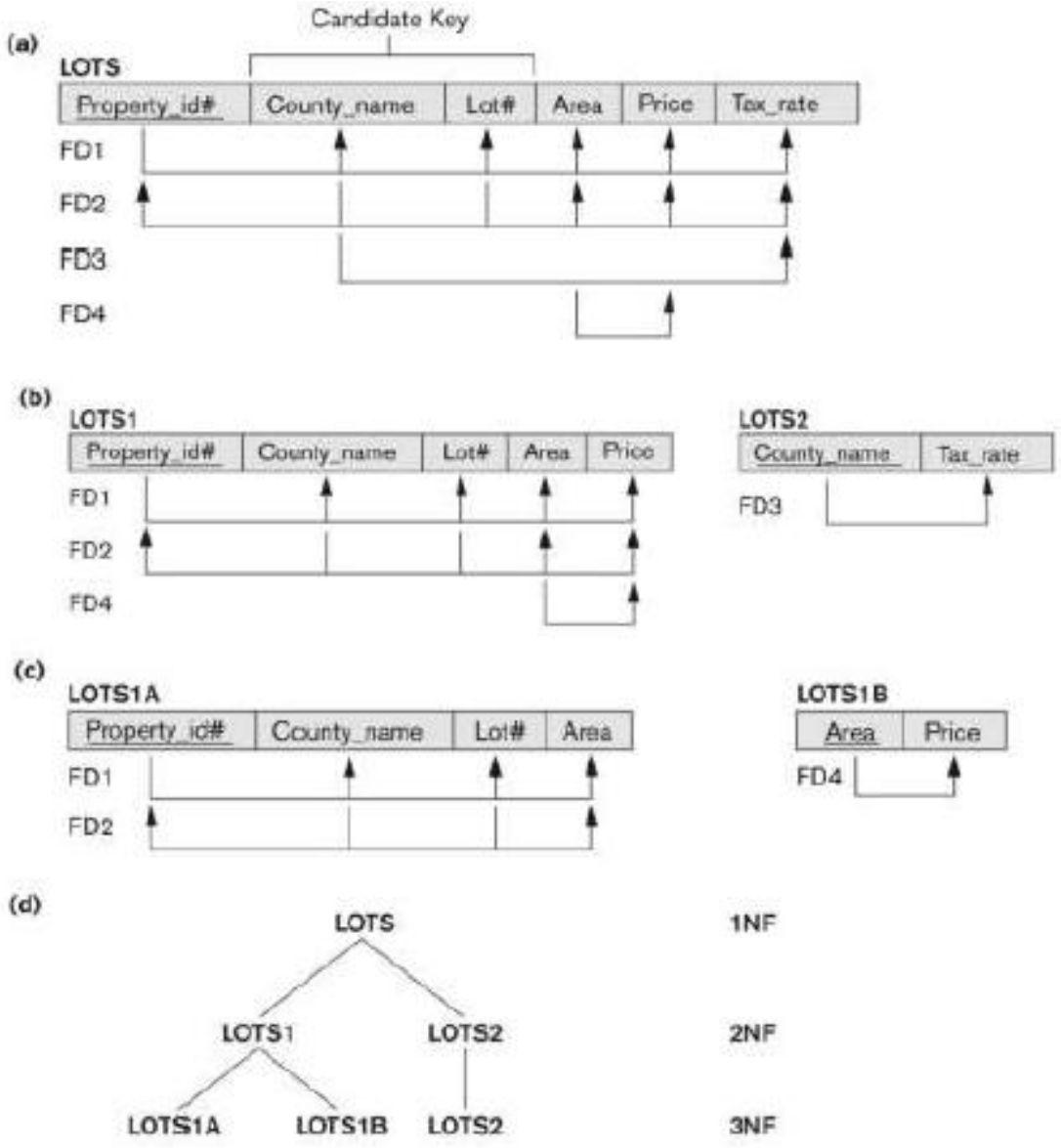


Figure
Normalization into 2NF and 3NF.
(a) The LOTS relation with its functional dependencies FD1 through FD4.
(b) Decomposing into the 2NF relations LOTS1 and LOTS2.
(c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B.
(d) Progressive normalization of LOTS into a 3NF design.

BCNF (Boyce-Codd Normal Form)

- A relation schema R is in Boyce-Codd Normal Form (BCNF) if whenever an FD X → A holds in R, then X is a superkey of R

- Each normal form is strictly stronger than the previous one
  - Every 2NF relation is in 1NF
  - Every 3NF relation is in 2NF
  - Every BCNF relation is in 3NF

- There exist relations that are in 3NF but not in BCNF

- Hence BCNF is considered a stronger form of 3NF

- The goal is to have each relation in BCNF (or 3NF)

Boyce-Codd normal form

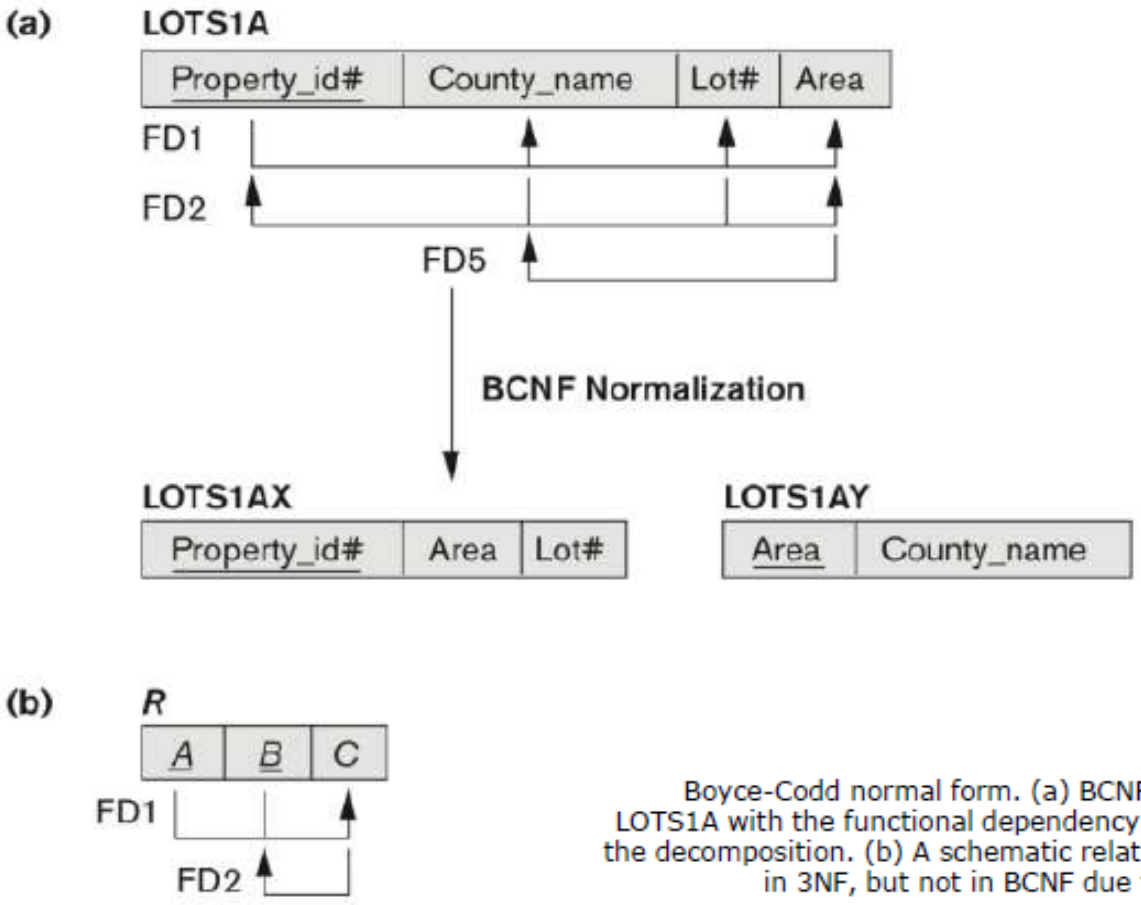


Figure
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF due to the f.d. C → B.

Lossless Join Decomposition

- Consider there is a relation R which is decomposed into sub relations R_1 , R_2 , …. , R_n.
- This decomposition is called lossless join decomposition when the join of the sub relations results in the same relation R that was decomposed.
- For lossless join decomposition, we always have-

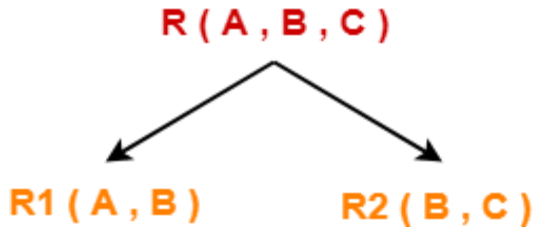
  $\mathbf{R_1 \bowtie R_2 \bowtie R_3 ……. \bowtie R_n = R}$ where $\bowtie$ is a natural join operator

Lossless Join Decomposition

- Consider the following relation R(A , B , C)-

A	B	C
1	2	1
2	5	3
3	3	3

R(A , B , C)



A	B
1	2
2	5
3	3

B	C
2	1
5	3
3	3

$R_1 \bowtie R_2 = R$

A	B	C
1	2	1
2	5	3
3	3	3

Lossless join decomposition is also known as **non-additive join decomposition.** This is because the resultant relation after joining the sub relations is same as the decomposed relation. No extraneous tuples appear after joining of the sub-relations.

Lossy Join Decomposition

- Consider there is a relation R which is decomposed into sub relations R_1 , R_2 , …. , R_n.

- This decomposition is called lossy join decomposition when the join of the sub relations does not result in the same relation R that was decomposed.

- The natural join of the sub relations is always found to have some extraneous tuples.

- For lossy join decomposition, we always have-

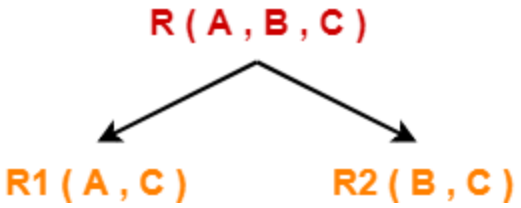
  $\mathbf{R_1 \bowtie R_2 \bowtie R_3 \ldots\ldots \bowtie R_n \supset R}$ where ⋈ is a natural join operator

Lossy Join Decomposition

- Consider the following relation R(A , B , C)-

A	B	C
1	2	1
2	5	3
3	3	3

R(A , B , C)



A	C
1	1
2	3
3	3

B	C
2	1
5	3
3	3

$R_1 \bowtie R_2 \supset R$

A	B	C
1	2	1
2	5	3
2	3	3
3	5	3
3	3	3

Lossy join decomposition is also known as **careless decomposition.** This is because extraneous tuples get introduced in the natural join of the sub-relations. Extraneous tuples make the identification of the original tuples difficult.

Decomposing Relations

- Consider a schema R(A, B, C, D) and functional dependencies A -> B and C -> D. Then the decomposition of R is : R1 (A, B) and R2(C, D)

- While decomposing a relational table we must verify the following properties:

i) **Dependency Preserving Prope**rty:

A decomposition is said to be dependency preserving if $F^+=(F1 \cup F2 \cup .. Fn)^+$, Where F^+=total functional dependencies(FDs) on universal relation R, F1 = set of FDs of R1, and F2 = set of FDs of R2.

For the above question R1 preserves A->B and R2 preserves C->D.

Since the FDs of universal relation R is preserved by R1 and R2, the decomposition is dependency preserving.

ii) **Lossless-Join Property:**

The decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies are in F+:-
a) R1 ∩ R2 -> R1
b) R1 ∩ R2 -> R2

It ensures that the attributes involved in the natural join ( ) are a candidate key for at least one of the two relations.

In the above question schema R is decomposed into R1 (A, B) and R2(C, D), and R1 ∩ R2 is empty. So, the decomposition is not lossless.

Exercise

- **Consider R = (A, B, C, D, E)**

  We decompose it into

  R1 = (A, B, C)

  R2 = (A, D, E)

  The set of functional dependencies are:

  $A \rightarrow BC$

  $CD \rightarrow E$

  $B \rightarrow D$

  $E \rightarrow A$

Show that this decomposition is a lossless-join decomposition.

Solution

- **R = (A, B, C, D, E)**

Given,

R1 = (A, B, C)

R2 = (A, D, E)

R1 ∩ R2 = A

Given FD:

(A → BC)

(A → ABC)

(R1 ∩ R2 → R1)

Hence this is a lossless-join decomposition.

Example

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

A relation TEACH that is in 3NF but not in BCNF

Achieving the BCNF by Decomposition (1)

- Two FDs exist in the relation TEACH:
  - fd1: { student, course} ->instructor
  - fd2: instructor ->course

- {student, course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure (b).
  - So this relation is in 3NF *but not in BCNF*

- A relation **NOT in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations.**

Achieving the BCNF by Decomposition (2)

- Three possible decompositions for relation TEACH
  - D1: {student, instructor} and {student, course}
  - D2: {course, instructor} and {course, student}
  - D3: {instructor, course } and {instructor, student}

- All three decompositions will lose fd1.
  - We have to settle for sacrificing the functional dependency preservation. But we cannot sacrifice the non-additivity property after decomposition.

- Out of the above three, only the 3rd decomposition will not generate spurious tuples after join(and hence has the non-additivity property))
  - R1 ∩ R2 -> R1

Differences between 3NF and BCNF

BASIS FOR COMPARISON	3NF	BCNF
Concept	No non-prime attribute must be transitively dependent on the Candidate key.	For any trivial dependency in a relation R say X->Y, X should be a super key of relation R.
Dependency	3NF can be obtained without sacrificing all dependencies.	Dependencies may not be preserved in BCNF.
Decomposition	Lossless decomposition can be achieved in 3NF.	Lossless decomposition is hard to achieve in BCNF.

Note: Redundancy in BCNF is low when compared to 3NF.

Exercise

- For a relation R(A, B, C, D, E, F) with following function dependencies for the relation R:

AB -> CDE

D -> F

Decompose the relations to 3NF.

Solution

- R(A, B, C, D, E, F)

  AB -> CDE

  D -> F

- Observing functional dependencies, we can consider **AB** as a candidate key for relation R because using key AB we can search the value for all the attribute in a relation R.

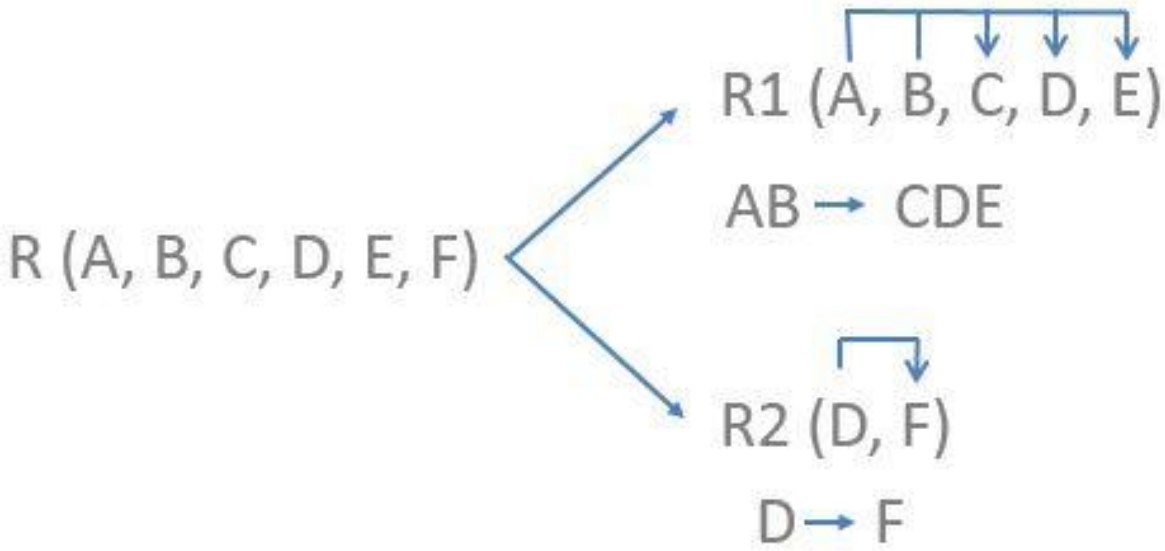
- So **A, B** becomes **prime attributes** as they together make candidate key.

- The attributes **C, D, E, F** becomes **non-prime** attributes because none of them is the part of a candidate key.

Solution contd....

- The table is in 2NF as no non-prime attribute is partially dependent on candidate key.
- But, a transitive dependency is observed among the functional dependencies provided, as the attribute **F** is not directly dependent on candidate key **AB**. Instead, attribute **F** is **transitively** dependent on candidate key **AB** via attribute **D**.
- Till attribute D has some value we can reach to attribute value of F, from the candidate key AB.
- In case the value of attribute D is NULL we can never find/search the value of F with the help of candidate key AB.
  - This is the reason why 3NF demands to remove the transitive dependency from the relations.

Solution contd....



- Now, the tables R1 and R2 are in 3NF as it has no partial and transitive dependencies left.
- Relation **R1 (A, B, C, D, E)** has a candidate key **AB** whereas, relation **R2 (D, F)** has **D** as its candidate key.

Exercise

- For a relation R(A, B, C, D, F) with following function dependencies for the relation R:

A -> BCDF

BF -> ACD

D -> F

Decompose the relation to BCNF.

Solution

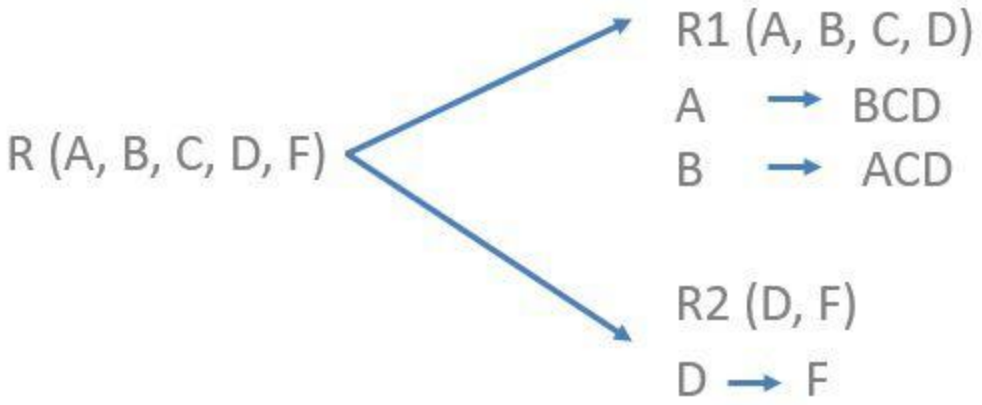
- By observing the relation R, we can say that **A** and **BF** are **candidate keys** of relation R, because they alone can search the value for all attributes in the relation R.

- So **A, B, F** are the **prime** attributes whereas, **C** and **D** are **non-prime** attributes.

- No transitive dependency is observed in the functional dependencies present above. Hence, the table R is in 3NF.

- But one functional dependency i.e. **D -> F** is violating the definition of BCNF, according to which, if D -> F exist then **D** should be the **super key** which is not the case here. So we will divide the relation R.

Solution contd….



- Now, the tables R1 and R2 are in BCNF.
- Relation **R1** has two **candidate keys A** and **B**, the trivial functional dependency of R1 i.e. A-> BCD and B -> ACD, hold for BCNF as A and B are the super keys for relation.
- Relation **R2** has **D** as its **candidate key** and the functional dependency D -> F also holds for BCNF as D is a Super Key.

Exercise

- The relation schema Student_Performance (name, courseNo, rollNo, grade) has the following FDs:

  name,courseNo->grade

  rollNo,courseNo->grade

  name->rollNo

  rollNo->name

Find the highest normal form of this relation schema.

Solution

- With the help of closure set of attributes we can find the candidate keys:
  - (name,courseNo)
  - (rollNo,courseNo)

- name->rollNo and rollNo->name are not partial functional dependencies, because for a functional dependency X->Y to be partially dependent, X should be prime attribute and Y should be non prime attribute. But here both X and Y are prime attributes. So the relation is in 2NF.

Solution contd....

- A **super key** is the set of attributes which can uniquely identify a tuple. A candidate key is a superkey. Adding zero or more attributes to candidate key generates super key.

- First two FDs (i.e. name,courseNo->grade & rollNo,courseNo->grade) satisfies first condition. Because name,courseNo & rollNo,courseNo is a superkey.

- Last two FDs (i.e name->rollNo & rollNo->name) satisfies second condition because rollNo and name is a prime attribute. Therefore the relation is also in 3NF.

Solution contd....

- In the 2 FDs :

  name->rollNo , name is not a superkey

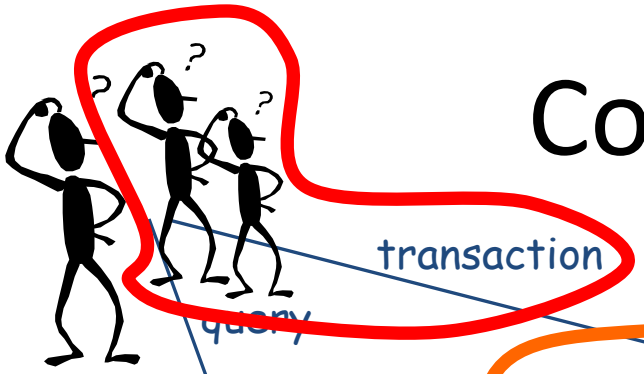
  rollNo->name, rollNo is not a superkey

  Hence the relation is not in BCNF.

  Therefore the highest normal form of the given relation schema is 3NF.

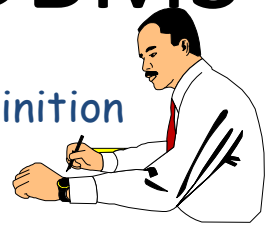
Transaction Management

Components of a DBMS



transaction

query



Data Definition

Query Compiler

Transaction Manager

Schema Manager

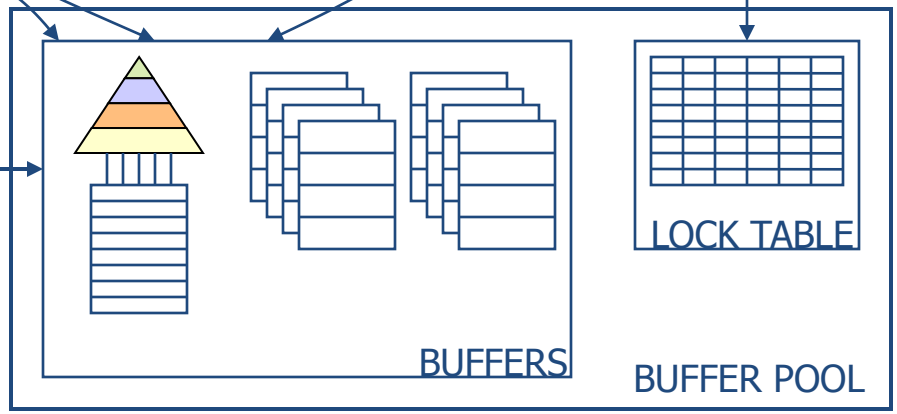
Execution Engine

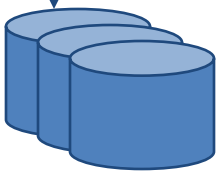
Logging/Recovery

Concurrency Control

Buffer Manager

Storage Manager





DBMS: a set of cooperating software modules

Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.

- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.

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

Transaction Concept

- E.g. transaction to transfer Rs.50 from account A to account B:

  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Concurrency Control & Recovery

- Very valuable properties of DBMSs
  - without these, DBMSs would be much less useful

- Based on concept of transactions with ACID properties

Statement of Problem

- Concurrent execution of independent transactions
  - utilization/throughput (“hide” waiting for I/Os.)
  - response time
  - fairness
- Example:

	T1:	T2:
t0:	tmp1 := read(X)	
t1:		tmp2 := read(X)
t2:	tmp1 := tmp1 – 20	
t3:		tmp2 := tmp2 + 10
t4:	write tmp1 into X	
t5:		write tmp2 into X

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issues:* Effect of *interleaving* transactions, and *crashes*.

Statement of problem (cont.)

- Arbitrary interleaving can lead to
  - Temporary inconsistency (ok, unavoidable)
  - “Permanent” inconsistency (bad!)
- Need formal correctness criteria.

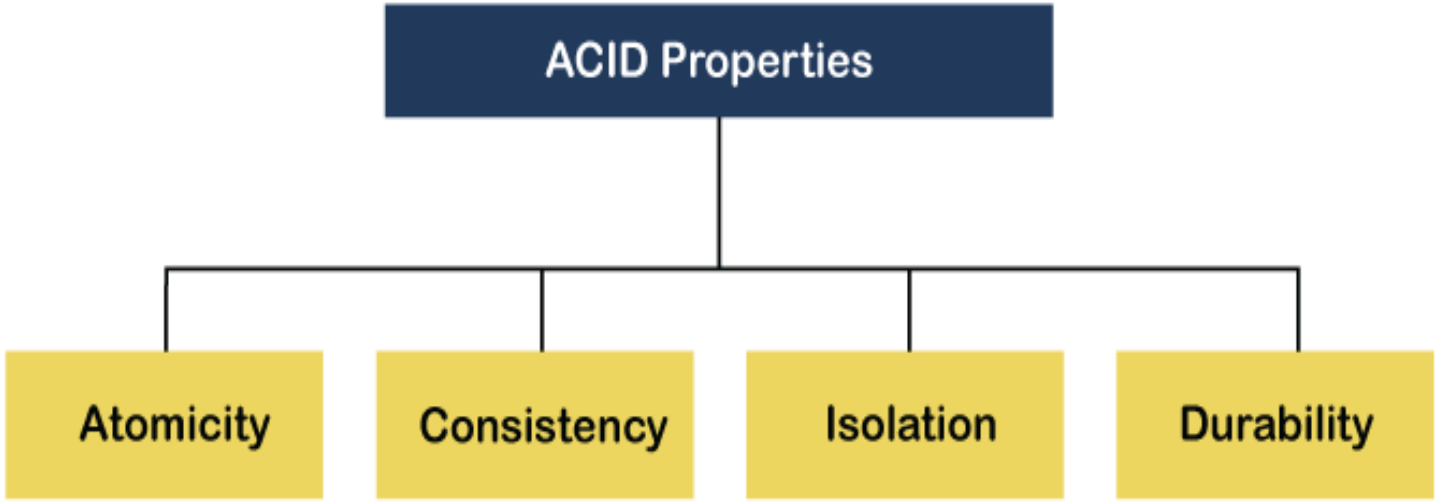
Definitions

- A program may carry out many operations on the data retrieved from the database
- However, the DBMS is only concerned about what data is read/written from/to the database.
- *database* - a fixed set of named data objects *(A, B, C, …)*
- *transaction* - a sequence of read and write operations *(read(A), write(B), …)*
  - DBMS's abstract view of a user program

ACID properties

- A **transaction** is a single logical unit of work which accesses and possibly modifies the contents of a database.
- Transactions access data using read and write operations.
  - Read is retrieving information from the database.
  - Write is inserting, updating, and deleting entries from the database.
- In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.

ACID properties contd..



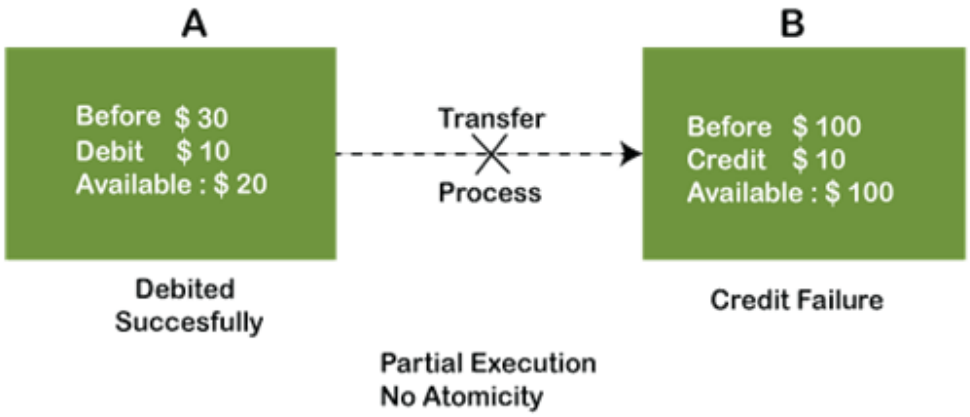
Atomicity

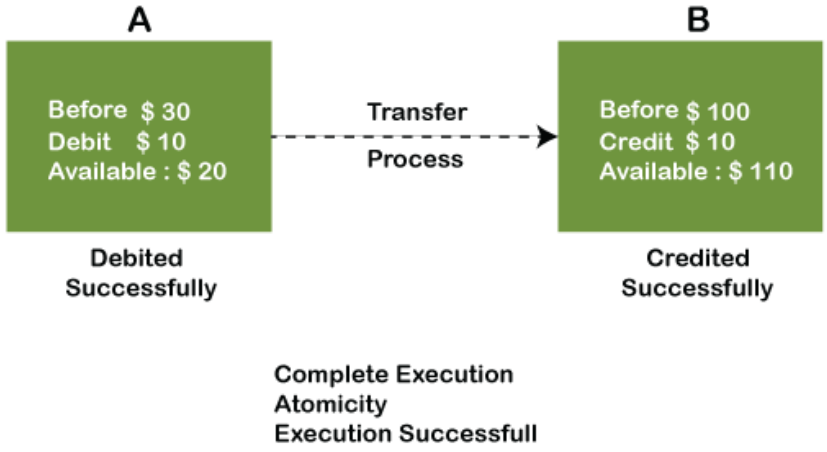
- It simply says “All or Nothing”. There is no intermediate.
- If there are any database transaction (set of the read/write operations), all the operations should be executed otherwise none.
- All the operation in the transaction is considered to be one unit or atomic task.
- If the system fails or any **read/write conflicts** occur during the transaction, the system needs to revert back to its previous state.
- It involves the following two operations.

  —**Abort**: If a transaction aborts, changes made to database are not visible.

  —**Commit**: If a transaction commits, changes made are visible.

Atomicity contd..

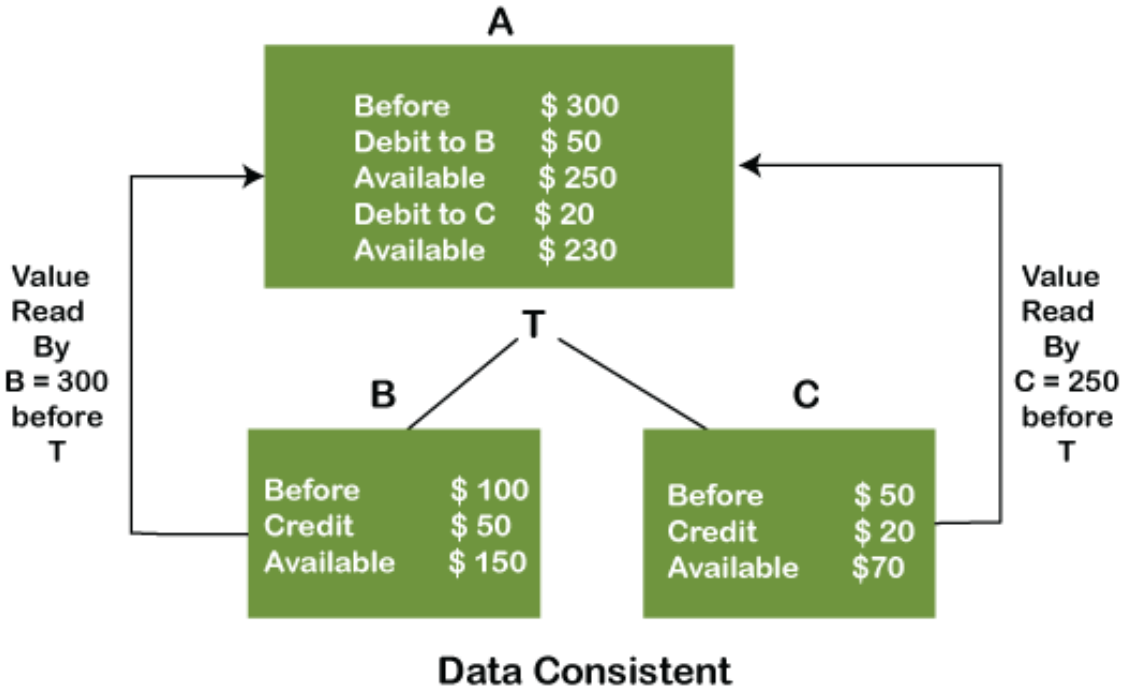




Consistency

- The word **consistency** means that the value should remain preserved always.
- Every attribute in the database has some rules to ensure the stability of the database.
- In DBMS, the integrity of the data should be maintained, which means if a change in the database is made, it should remain preserved always.
  - Integrity constraints must be maintained so that the database is consistent before and after the transaction.
- It refers to the correctness of a database.
  - The data should always be correct.
- A database is initially in a consistent state, and it should remain consistent after every transaction.

Consistency contd..



- The debit and credit operation from account A to B and C has been done successfully. We can see that the transaction is done successfully, and the value is also read correctly.
  - Thus, the data is consistent.

- In case the value read by B and C is $300, then it makes the data inconsistent because when the debit operation executes, it will not be consistent.

Isolation

- The term 'isolation' means separation.

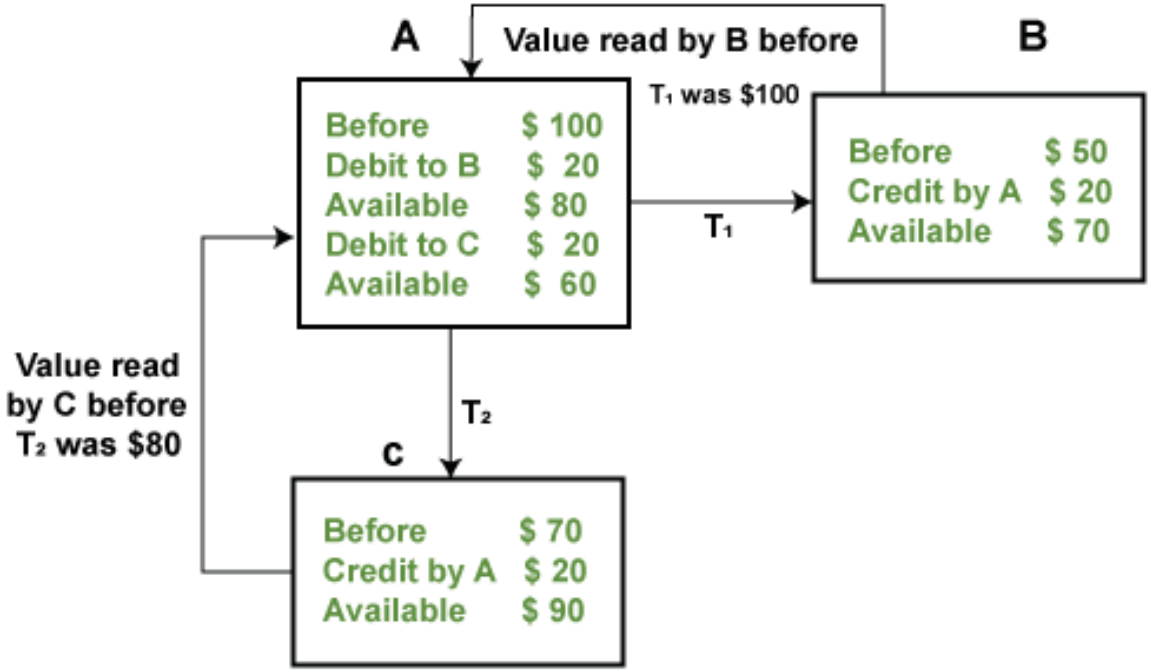
- Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.

  – This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

- If the multiple transactions are running concurrently, they should not be affected by each other

  i.e., the result should be the same as the result obtained if the transactions were running sequentially.

Isolation contd..



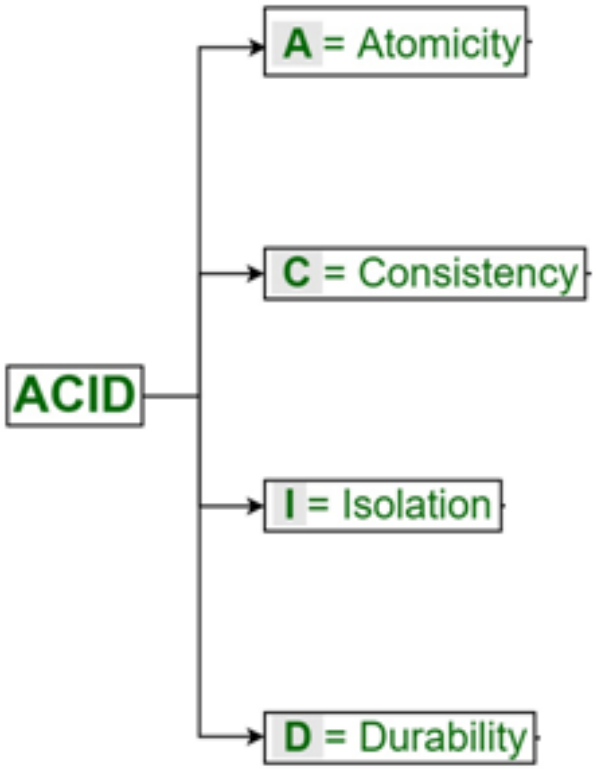
Isolation - Independent execution of T_1 & T_2 by A

- account A is making T1 and T2 transactions to account B and C, but both are executing independently without affecting each other. It is known as Isolation.

Durability

- Durability ensures the permanency of something.
- This is the **ACID Property After Completion of Transaction**.
- The term durability ensures that the data after the successful execution of the operation becomes permanent in the database.
- Changes that have been committed to the database should remain even in the case of software and hardware failure.
  - The durability of the data should be so perfect that even if the system fails or leads to a crash, the database still survives.
    - The information/data contained in the database should not disappear upon hardware or software failure.
- Example:
  - It may happen that a system gets crashed after completion of all the operations. If the system restarts it should preserve the stable state.
  - An amount in A and B's account should be the same before and after the system restart.

Summary: ACID properties



The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably/persistently stored.

ACID properties in DBMS make the transaction over the database more reliable and secure.

Therefore, the ACID property of DBMS plays a vital role in maintaining the consistency and availability of data in the database.

TRANSACTIONS AND SCHEDULES

- A transaction is seen by DBMS as sequence of reads and writes
  - read of object O denoted R(O)
  - write of object O denoted W(O)
  - must end with Abort or Commit

- A **schedule is a list of actions (reading, writing, aborting, or committing)** from a set of transactions, and the order in which two actions of a transaction *T appear in a schedule must be the same as the order in which they appear in T.*

  - A schedule of a set of transactions is a list of all actions where order of two actions from any transaction must match order in that transaction.

- Intuitively, a schedule represents an actual or potential execution sequence.

A schedule Example

T1: Transfer	T2: Interest
A=A+100	
	A=1.06*A
	B=1.06*B
B=B-100	

T1: Transfer	T2: Interest
Read(A)	
Write(A)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)
Read(B)	
Write(B)	

Complete schedule

- A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a **complete schedule**.

Complete Schedule

T_1	T_2
R(A)	
	R(B)
W(A)	
	W(B)
Commit	
	Abort

Complete Schedule

T_1	T_2
R(A)	
W(A)	
	R(B)
Commit	
	W(B)
	Abort

Complete Schedule

T_1	T_2
R(A)	
W(A)	
Commit	
	R(B)
	W(B)
	Abort

- A complete schedule must contain all the actions of every transaction that appears in it.

- **Note:** Consequently, a complete schedule wilt not contain any active transaction at the end of the schedule.

Serial schedule

- If the actions of different transactions are not interleaved that is, transactions are executed from start to finish, one by one then the schedule is a **serial schedule**.

Serial Schedule

T_1	T_2
	$R(A)$
	$W(A)$
$R(B)$	
$W(B)$	

Serial Schedule

T_1	T_2
$R(B)$	
$W(B)$	
	$R(A)$
	$W(A)$

- A **serial schedule** is a schedule in which the different transactions are not interleaved (i.e., transactions are executed from start to finish one-by-one).

CONCURRENT EXECUTION OF TRANSACTIONS

- The DBMS interleaves the actions of different transactions to improve performance
  - but not all interleavings should be allowed.

- Ensuring transaction isolation while permitting concurrent execution is difficult but necessary for performance reasons:

  - First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction.
    - Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle and increases system throughput (the average number of transactions completed in a given time).

  - Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly.

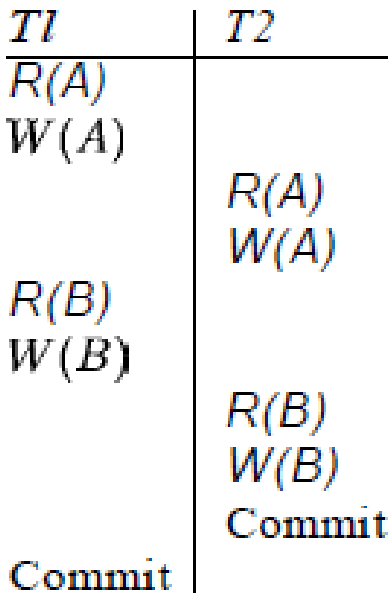
Serializable Schedule

- A **serializable schedule over a set *S of committed transactions is a schedule*** whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over *S.*
  - *That is, the database instance* that results from executing the given schedule is identical to the database instance that results from executing the transactions in *some serial order.*

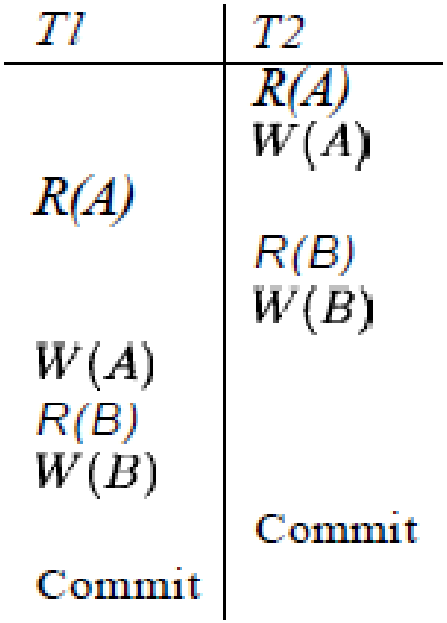
- *In short a* **serializable schedule** *is a* schedule that is equivalent to some serial execution of the transactions.

- Executing transactions serially in different orders may produce different results, but all are presumed to be acceptable.
  - the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution.

Serializable Schedule : Examples



T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	



T1	T2
	R(A)
	W(A)
R(A)	
	R(B)
	W(B)
W(A)	
R(B)	
W(B)	
	Commit
Commit	

T1 's read and write of *B is not influenced by T2's actions on A, and the net effect is the same* if these actions are 'swapped' to obtain the serial schedule *T1 ; T2.*

The left two example transactions from Figure can also be interleaved as shown here. This schedule, also serializable, is equivalent to the serial schedule *T2; T1.*

When can actions be re-ordered?

- Let I, J be two consecutive actions of T1 and T2
  - I=Read(O), J=Read(O)
  - I=Read(O), J=Write(O)
  - I=Write(O), J=Read(O)
  - I=Write(O), J=Write(O)
- If I and J are both reads, then they can be freely reordered.
- In all other cases, order impacts outcome of schedule.

Non-serializable schedule

- A DBMS might sometimes execute transactions in a way that is not equivalent to any serial execution; that is, using a schedule that is not serializable.

- This can happen for two reasons:

  – First, the DBMS might use a concurrency control method that ensures the executed schedule, though not itself serializable, is equivalent to some serializable schedule.

  – Second, SQL gives application programmers the ability to instruct the DBMS to choose non-serializable schedules

Formal Properties of Schedules

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.

- *Equivalent schedules*: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.

- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

  (Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with Interleaved Execution

- Not all interleavings of operations are okay.
- **Anomaly: two consistency-preserving** committed transactions that lead to an inconsistent state.
- Two actions on the same data object conflict if at least one of them is a write.
- Types of anomalies:
  - Reading Uncommitted Data (WR Conflicts) - “dirty reads”
  - Unrepeatable Reads (RW Conflicts)
  - Overwriting Uncommitted Data (WW Conflicts)

Reading Uncommitted Data (WR Conflicts)

- "Dirty Read"
  - The first source of anomalies is that a transaction *T2 could read a database* object *A that has been modified by another transaction Tl, which has not yet* committed. Such a read is called a dirty read.

- Inconsistent result of A is exposed to transaction T2

T1: Transfer	T2: Interest
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Unrepeatable Reads (RW Conflicts)

- T1 could see two values for A, although it has not changed A itself.

T1	T2
R(A)	
	R(A)
	W(A)
	Commit
R(A)	
W(A)	
Commit	

Overwriting Uncommitted Data (WW Conflicts)

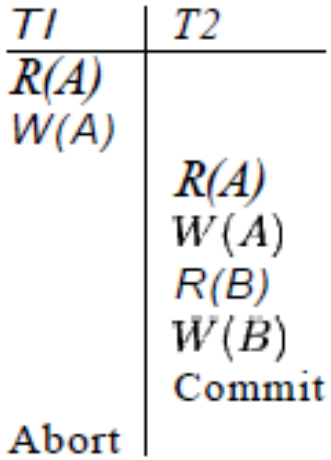
- The problem is that we have a **lost update.**

T1	T2
	W(A)
W(B)	
	W(B)
	Commit
W(A)	
Commit	

Schedules Involving Aborted Transactions

- A **serializable schedule over a set *S of transactions is a schedule whose effect on*** any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of *committed transactions in S.*

- This definition of serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations.



Concurrency control schemes

- The DBMS must provide a mechanism that will ensure all possible schedules are:
  - Serializable
  - Recoverable, and preferably cascadeless
- Concurrency control protocols ensure these properties.

How do we ensure Serializability

- This is the task of the scheduler.

- There are two basic techniques:
  - Locking
  - Time-Stamp Ordering

- Locking enforces serializability by ensuring that no two transactions access conflicting objects in an “incorrect” order.
  - A DBMS must be able to ensure that only serializable, recoverable schedules are allowed and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a *locking protocol to achieve* this.

- Time-Stamp ordering assigns a fixed order for every pair of transactions and ensures that conflicting accesses are made in that order.

Lock-Based Concurrency Control

- Lock - associated with some object
  - shared or exclusive
- Lock and unlock requests are handled by the lock manager
- Locking protocol - set of rules to be followed by each transaction to ensure good properties.
- In Lock-Based Protocol , any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

- It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

2. Exclusive lock:

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

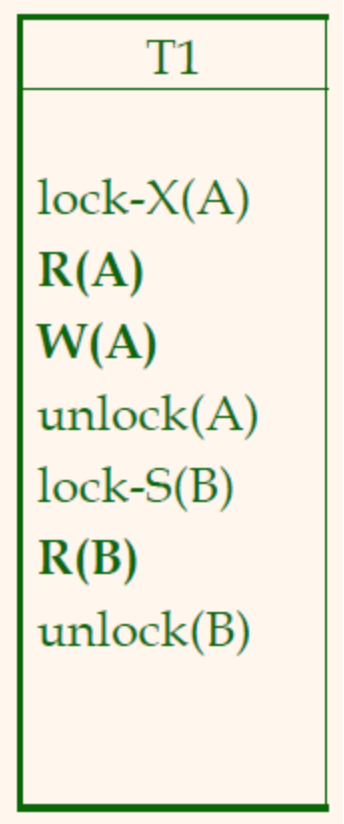
Lock Compatibility Matrix

- Locks on a data item are granted based on a lock compatibility matrix:

		Mode of Data Item		
		None	Shared	Exclusive
Request mode	Shared	Y	Y	N
	Exclusive	Y	N	N

- When a transaction requests a lock, it must wait (block) until the lock is granted.
- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Transaction performing locking



Two-Phase Locking (2PL)

- Two-Phase Locking Protocol
  - Each transaction must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - *A transaction can release its locks once it has performed its desired operation (R or W). A transaction cannot request additional locks once it releases any locks.*
  - If an transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object.
- Note: locks can be released before transaction completes (commit/ abort). 2PL starts with a “growing” phase, where locks are requested followed by a “shrinking” phase, where locks are released

Two-Phase Locking (2PL) Example

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	——	——
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	——	——

The following way shows how unlocking and locking work with 2-PL.

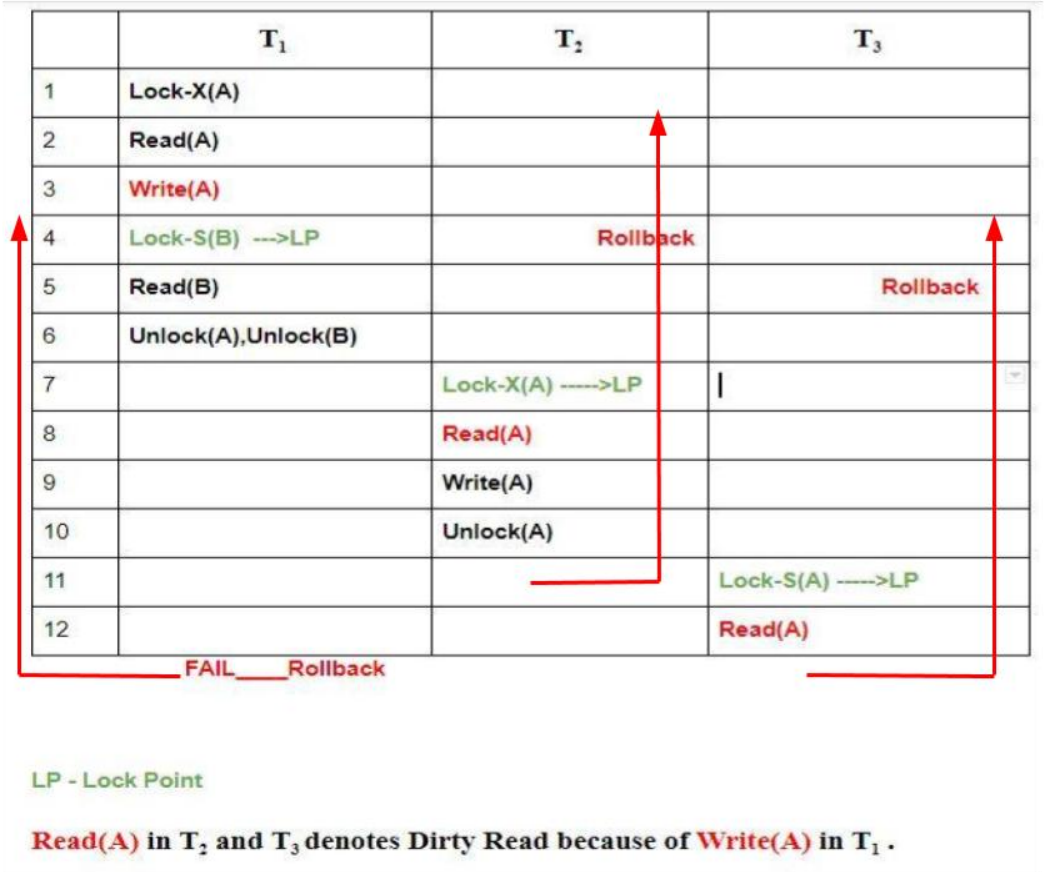
Transaction T1:
Growing phase: from step 0-2
Shrinking phase: from step 5-6
Lock point: at 3

Transaction T2:
Growing phase: from step 1-5
Shrinking phase: from step 7-8
Lock point: at 6

What is **LOCK POINT ?**
The Point at which the growing phase ends, i.e., when transaction takes the final lock it needs to carry on its work.

Drawbacks of 2-PL (1)

- Cascading Rollback



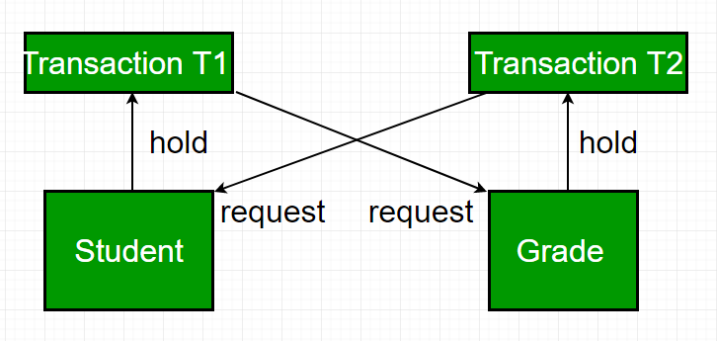
	T_1	T_2	T_3
1	Lock-X(A)		
2	Read(A)		
3	Write(A)		
4	Lock-S(B) --->LP	Rollback	
5	Read(B)		Rollback
6	Unlock(A),Unlock(B)		
7		Lock-X(A) ----->LP	
8		Read(A)	
9		Write(A)	
10		Unlock(A)	
11			Lock-S(A) ----->LP
12			Read(A)

LP - Lock Point

Read(A) in T_2 and T_3 denotes Dirty Read because of Write(A) in T_1 .

Drawbacks of 2-PL (2)

- Deadlocks and Starvation
  - In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks.



  - Starvation is the situation when a transaction has to wait for a indefinite period of time to acquire a lock.

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Relatively rare schedules lead to deadlock
- The DBMS must either prevent or detect (and resolve) such deadlock situations
- The common approach is to detect and resolve deadlocks.
  - A simple way to identify deadlocks is to use a timeout mechanism.

Strict Two-Phase Locking (Strict 2PL)

- The most widely used locking protocol, called *Strict Two-Phase Locking, or Strict 2PL, has two rules:*

1. If a transaction *T wants to read (respectively, modify) an object, it* first requests a shared (respectively, exclusive) lock on the object.

2. All locks held by a transaction are released when the transaction is completed.

OR

- Each transaction must obtain a S (*shared) lock on object* before reading, and an X (*exclusive) lock on object* before writing.

- All X (exclusive) locks acquired by a transaction must be held until completion (commit/abort).

- A transaction can not request additional locks once it releases any locks.

Strict Two-Phase Locking (Strict 2PL) Example

T1	T2
X(A)	
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

Schedule Illustrating Strict 2PL with Serial Execution

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	X(B)
	R(B)
	W(B)
	Conllnit
X(C)	
R(C)	
W(C)	
Commit	

Schedule Following Strict 2PL with Interleaved Actions

Difference between 2PL and Strict 2PL

- The difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
  - Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.
- Strict 2PL allows only serializable schedules.

TRANSACTION SUPPORT IN SQL

- Consider what support SQL provides for users to specify transaction-level behavior.

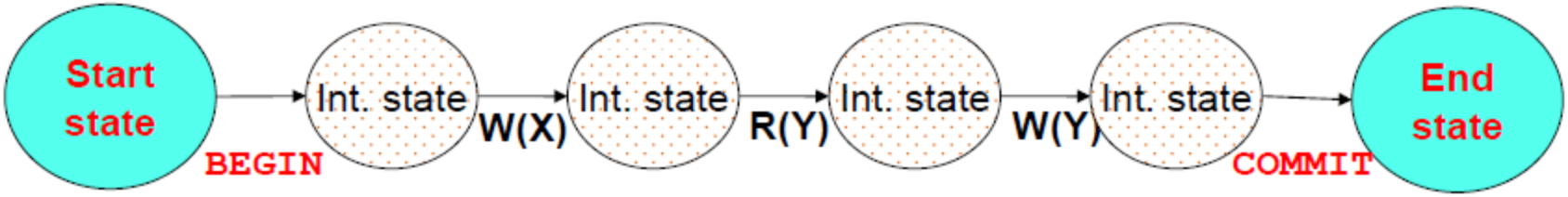
- **Creating and Terminating Transactions**

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

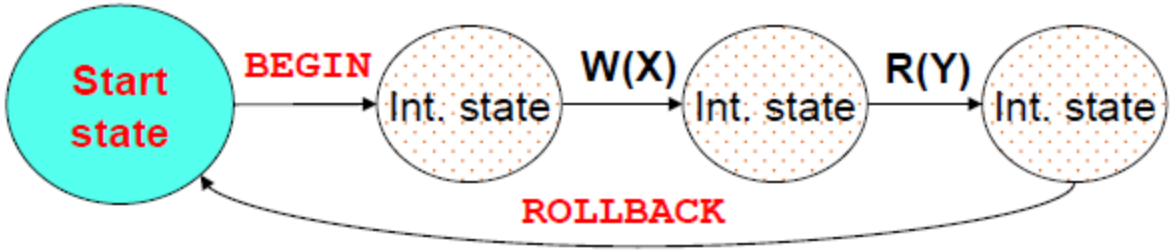
  – Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either a COMMIT command or a ROLLBACK (the SQL keyword for abort) command.

Transaction model

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



- If, for some reason, the transaction is unable to complete successfully, the DBMS should “undo” any change possibly made to the DB

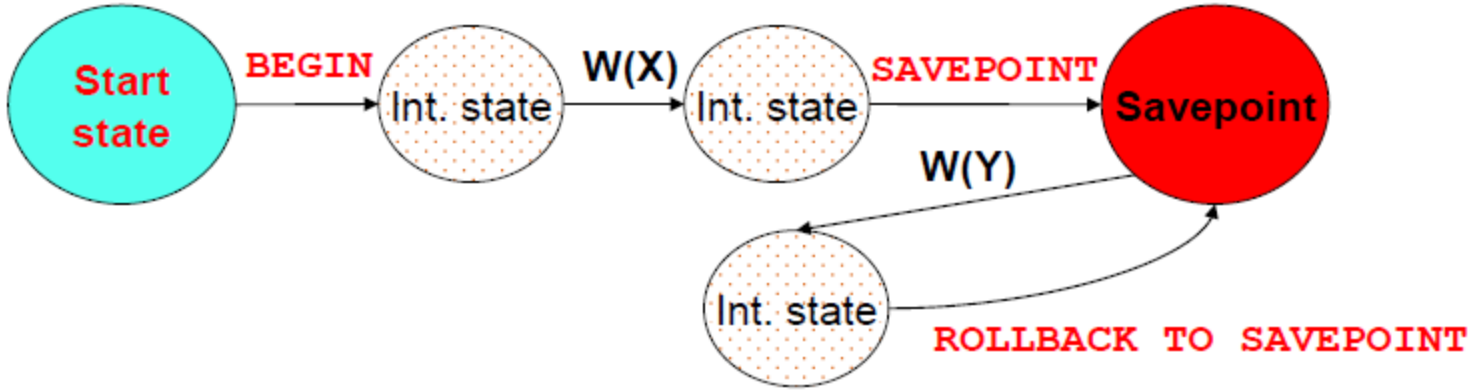


Transactions Features to support applications

- Features to support applications that involve long-running transactions, or that must run several transactions one after the other:
  - Savepoint
  - Chained transactions
- Savepoint : allows to identify a point in a transaction and selectively roll back operations carried out after this point.
  - This is especially useful if the transaction carries out what-if kinds of operations, and wishes to undo or keep the changes based on the results.
  - The savepoint command to give each savepoint a name:
    - SAVEPDINT *(savepoint name)*
  - A subsequent rollback command can specify the savepoint to roll back to
    - ROLLBACK TO SAVEPDINT *(savepoint name)*

Transactions with savepoint

- The transaction model used by DBMS is actually more complex; in particular, it is possible to define some “savepoint”, which can be used to undo the operations of a transaction only partially.



- The savepoint mechanism offers two advantages:
  - can roll back over several savepoints.
    - can roll back only the most recent transaction, which is equivalent to rolling back to the most recent savepoint,
  - the overhead of initiating several transactions is avoided.

Chained transactions

- Commit or roll back a transaction and immediately initiate another transaction.

- This is done by using the optional keywords AND CHAIN in the COMMIT and ROLLBACK statements.

What to Lock?

- The DBMS can lock objects at different **granularities:**
  - can lock entire tables or set row-level locks

- **Phantom problem: A transaction retrieves** a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself.

- To prevent phantoms, the DBMS must conceptually lock *all possible rows on behalf of transaction.*

  - *lock the entire table, at* the cost of low concurrency

  - It is possible to take advantage of indexes to do better

Transaction Characteristics in SQL

- SQL allows them to specify three characteristics of a transaction:
  - access mode
  - diagnostics size and
  - isolation level

- The **diagnostics** size determines the number of error conditions that can be recorded.

- If the access **mode is**

  - **READ ONLY - the transaction is not allowed to modify** the database.
    - Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. Only shared locks need to be obtained, thereby increasing concurrency

  - **READ WRITE - the transaction is allowed to modify** the database
    - INSERT, DELETE, UPDATE, and CREATE commands can be executed.

Transaction Characteristics in SQL contd..

- The **isolation level controls the extent to which a given transaction is exposed** to the actions of other transactions executing concurrently.

- Isolation level choices are
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ and
  - SERIALIZABLE

- The effect of these levels is summarized in Figure. In this context, *dirty read and unrepeatable read are defined as usual.*

Transaction Characteristics in SQL contd..

- Isolation levels

Level	Dirty Read	Unrepeatable Read	
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Isolation levels

- REPEATABLE READ ensures that *T reads only the changes made by committed* transactions and no value read or written by *T is changed by any other* transaction until *T is complete.*
  - *T could experience the phantom* phenomenon;
  - A REPEATABLE READ transaction sets the same locks as a SERIALIZABLE transaction, except that it does not do index locking; that is, it locks only individual objects, not sets of objects.
- READ COMMITTED ensures that *T reads only the changes made by committed* transactions, and that no value written by *T is changed by any other transaction* until *T is complete.*
  - A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete.
  - *T could experience the phantom* phenomenon
- A READ UNCOMMITTED transaction *T can read changes made to an object by an* ongoing transaction; obviously, the object can be changed further while *T is in* progress, and *T is also vulnerable to the phantom problem.*
  - A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions

Isolation levels contd..

- The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions.
  - The highest degree of isolation from the effects of other transactions is achieved by setting the isolation level for a transaction *T to SERIALIZABLE.*
  - *This isolation* level ensures that *T reads only the changes made by committed transactions.*
  - A SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged and holds them until the end, according to Strict 2PL.
  - Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance.

Transaction Characteristics in SQL contd..

- The isolation level and access mode can be set using the SET TRANSACTION command.
- For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:
  - SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
- When a transaction is started, the default is SERIALIZABLE and READ WRITE.

Thank you