DBMS ESE

23 November 2023 08:12 PM

K is a **superkey** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*

Superkey *K* is a **candidate key** if *K* is minimal

Foreign key constraint: Value in one relation must appear in another

- **Referencing** relation
- **Referenced** relation

Relational algebra is a procedural language

- Consists of 6 basic operations
  - Projection
  - Selection
  - Union
  - Cross Product
  - Rename
  - Set Difference
- Derived operations
  - Join
  - Intersection
  - Division

$\Pi_{A,C}(r)$

PROJECTION

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

$\sigma_{A=B \wedge D > 5}(r)$

SELECTION

A	B	C	D
α	α	1	7
β	β	23	10

Performs union between two given relations

- **r**, and **s** must have the same number of attributes
- Attribute domains must be compatible
- Duplicate tuples are automatically eliminated

Relations *r, s:*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

$r \cup s$:

A	B
α	1
α	2
β	1
β	3

FOR SET DIFFERENCE

Conditions

- No of attributes must be same
- Domain of attributes must be compatible

INTERSECTION

- $r \cap s$

A	B
α	2

Note: $r \cap s = r - (r - s)$

- Conditions
  - No of attributes must be same
  - Domain of attributes must be compatible

Renaming a Table

- Allows us to refer to a relation, (say E) by more than one name.

$\rho_x(E)$

returns the expression *E* under the name *X*

$r \times \rho_s(r)$

r.A	r.B	s.A	s.B
α	1	α	1
α	1	β	2
β	2	α	1
β	2	β	2

Question: Find the maximum value of B

- $\Pi_{r.B}$ - $\Pi_{r.B}(\sigma_{r.B<s.B}(r \times \rho_s(r)))$

B values that are lesser than some B values

Division operator A÷B (or A/B) can be applied if

- **Attributes of B is proper subset of Attributes of A**

The relation returned by division operator will return those **tuples from relation A which are associated to every B's tuple**

- The relation returned by division operator will have attributes = (All attributes of A – All Attributes of B)

The relation returned by division operator will return those **tuples from relation A which are associated to every B's tuple**

Retrieve *Sid* of students who enrolled in *every (or all)* course

- E(Sid,Cid)/C(Cid)
- $\Pi_{sid}(E) - \Pi_{Sid}(\Pi_{Sid}(E) \times \Pi_{Cid}(C) - E)$

Students **Sid** who are not enrolled in at least one course: Disqualified Sid

Sid	Cid
S1	C1
S2	C1
S1	C2
S3	C2

Enrolled E

Cid
C1
C2

Course C

Types of Joins

- Cross Join Basic join
- Natural Join
- Self Join
- Theta Join or Condition Join
- Equi Join
- Outer Join

Natural Join

Let *r* and *s* be relations on schemas *R* and *S* respectively. Then, the "natural join" of relations *R* and *S* is a relation on schema $R \cup S$ obtained as follows:

- Consider each pair of tuples t_r from *r* and t_s from *s*.
- If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple *t* to the result, where
  - *t* has the same value as t_r on *r*
  - *t* has the same value as t_s on *s*

Perform a Natural Join only if there is at least one common attribute that exists between two relations

Natural Join

- $r \bowtie s$

Theta Join

- Theta join combines tuples from different relations provided they satisfy the theta condition
  - The join condition is denoted by the symbol **θ**
  - Theta join can use **all kinds of comparison operators**
  - Denoted as $r \bowtie_\theta s$

Equi Join

Equi joins combine tables based on matching values in specified columns

- When Theta join uses only **equality** comparison operator, it is said to be Equi join
- Need to specify column

Outer Joins

Theta Join, Equi Join, and Natural Join are called inner joins

- Includes only those tuples with matching attributes and the rest are discarded in the resulting relation

Outer Join

- Include all the tuples from at least one participating relation

Left Outer Join

- Gives matching rows (similar as Natural Join) and rows which are in left table but not in right table

Right Outer Join

- Gives matching rows (similar as Natural Join) and rows which are in right table but not in left table

Full Outer Join

- Left Outer Join ∪ Right Outer Join

MODIFICATION OPERATIONS

Insertion is expressed in relational algebra by

- $r \leftarrow r \cup E$
  - Where r is a relation and E is a relational algebra expression

Example

- Insert tuple with $1200 in account A-973 at the Perryridge branch.
  - account ← account ∪ {("Perryridge", A-973, 1200$)}

A deletion is expressed in relational algebra by

- $r \leftarrow r - E$
  - Where r is a relation and E is a relational algebra expression

Example

- Delete all account records in the Perryridge branch
  - account ← account – $\sigma_{\text{branch-name = "Perryridge"}}$ (account)

Use the generalized projection operator to do this task

- $r \leftarrow \prod_{F1, F2, \ldots, Fl} (r)$

Make interest payments by increasing all balances by 5 percent

- account ← $\prod_{\text{AN, BN, BAL * 1.05}}$ (account)
  - Where AN, BN and BAL stand for account-number, branch-name and balance, respectively

AGGREGATE FUNCTIONS

Find minimum salary

- $F_{\text{MIN (Salary)}}$ (EMP)

Find average salary

- $F_{\text{AVG (Salary)}}$ (EMP)

Count names

- $F_{\text{Count (Name)}}$ (EMP)

Anomaly

Insertion Anomaly

- If we want to add new department information
  - We cannot insert because we need primary key ID

Deletion Anomaly

- Deletion of some data would force to delete other data
- If we remove only faculty in department
  - We will lose department information
  - For Example, if we remove faculty 98345, Kim in the previous table

Updation Anomaly

- All redundant copies has to be updated

Decomposition should be **lossless** and **dependency preserving**

Normalization is a method to reduce or remove redundancy from a table

- Redundancy occurs in a table or relation if two or more independent relations stored in a single table

Normalization reduces or removes insertion, deletion, and updation anomaly

- Decomposition helps to achieve (maintain) normalization

Redundancy in a relation may cause insertion, deletion and updation anomalies

Normal forms are used to eliminate or reduce redundancy in **database** tables

Normalization is the process of minimizing **redundancy** from a relation or set of relations

FIRST NORMAL FORM

A relational schema R is in **first normal form** if the domains of all attributes of R are atomic

- **Relation should not have any multi valued attribute**

Domain is **atomic** if its elements are considered to be indivisible units

Non-atomic values complicate storage and encourage redundant (repeated) storage of data

Functional Dependency

Require that the value for a certain set of **attributes determines uniquely the value** for another set of attributes

- **Describe dependency or relationship between attributes**
- For Example, $x \rightarrow y$
  - Here, x determines y or y is determined by x
    - x is a **determinant** attribute and y is **dependent** attribute

The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on *R* if and only if for any legal relations *r*(R), whenever any two tuples t_1 and t_2 of *r* agree on the attributes α, they also agree on the attributes β. That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

Trivial Functional Dependency

- If $x \subseteq y$ then $y \rightarrow x$ is a trivial functional dependency (FD), where x and y are some attributes over relation R
  - **Always valid,** (RHS of FD is a subset of LHS of FD)
  - Example, Sid $\rightarrow$ Sid, SidSname $\rightarrow$ Sid

Non-trivial Functional Dependency

- If $x \cap y = \phi$ then $y \rightarrow x$ is a non trivial functional dependency
  - **Need to check validity**
  - Example, Sid $\rightarrow$ Sname

Reflexive rule: if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (trivial and valid)

Augmentation rule: if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$ (valid)

Transitivity rule: if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (valid)

Union rule: If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (valid)

Decomposition rule: If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (valid)

Pseudotransitivity rule: If $\alpha \rightarrow \beta$ holds and $\gamma\ \beta \rightarrow \delta$ holds, then $\alpha\ \gamma \rightarrow \delta$ holds (valid)

Composition rule: If $\alpha \rightarrow \beta$ holds and $\delta \rightarrow \gamma$ holds, then $\alpha\delta \rightarrow \beta\gamma$ holds (valid)

Given a set *F* set of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.

- If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- etc.

The **set of all functional dependencies logically implied by *F*** is the **closure** of *F*.

We denote the *closure* of *F* by F^+.

It **helps to find all candidate keys in a relation**

CANONICAL COVER

We can **reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set.**

This simplified set of functional dependency is termed the **canonical cover. It is irreducible set of functional dependency.**

A **canonical cover** for *F* is a set of dependencies F_c such that

- *F* logically implies all dependencies in F_c, and
- F_c logically implies all dependencies in *F*, and
- No functional dependency in F_c contains an extraneous attribute
  - An attribute of a functional dependency is said to be extraneous if we **can remove it without changing the closure of the set of functional dependencies**
- Each **left side of functional dependency in F_c is unique**. That is, there are no two dependencies in F_c
  - $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ such that
  - $\alpha_1 = \alpha_2$

To compute a canonical cover for *F*:

repeat

Use the union rule to replace any dependencies in *F* of the form

$\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\ \beta_2$

Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous attribute either in α or in β

If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$

until (F_c not change)

Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

MINIMAL COVER

Canonical Cover vs. Minimal Cover

- A canonical cover can have more than one attribute on the right hand side.
- A minimal cover cannot allow more than one attribute on the right hand side.
- Example: A -> BC can be canonical cover
  - Whereas the minimal cover for the same functional dependency would be A -> B, A -> C

Example: FD: {A -> B, C -> B, D -> ABC, AC -> D}

- Step 1. {A -> B, C -> B, D -> A, D -> C, D -> B, AC -> D}
- Step 2. {A -> B, C -> B, D -> A, D -> C, ~~D -> B~~, AC -> D}
- Step 3. {A -> B, C -> B, D -> A, D -> C, AC -> D}
  - A^+ is not able to derive C and C^+ is not able to derive A. We therefore cannot remove any attribute at LHS of AC -> D
- Final Minimal Cover: {A -> B, C -> B, D -> A, D -> C, AC -> D}
- Final Canonical Cover: {A -> B, C -> B, D -> AC, AC -> D}

Method to find Minimal Cover

- First, make RHS of each FD as a single attribute or decompose RHS
  - Use **decomposition rule**
- Second, pick each functional dependency (FD) and check if other functional decencies (excluding current FD) can generate the same results/dependencies
  - If other FDs can generate, remove the current FD
    - Find the closure of LHS of each FD from other FDs (excluding current FD) and check if other dependencies can generate RHS of current FD. If so, remove current FD
  - This step is called **removal of redundant FD**

Method to find Minimal Cover

- Third, pick those FDs that have at least 2 attributes at LHS and try to reduce it (preferably make it one attribute on the LHS). E.g. xy->z
  - Check if closure of one attribute (x) at LHS can derive other attribute (y) of LHS. If so, remove other attribute (y)
    - Similarly we can check if x can be removed
  - This step is called **removal of extraneous attribute**

SECOND NORMAL FORM

Second Normal Form

Conditions for second normal form (2nd NF):

- Relation must be in 1st normal form and
- All non-prime attributes should be fully functional dependent on candidate key
  - Or Relation **should not contain any partial dependency**
    - Partial Dependency occurs when a non-prime attribute is functionally dependent on part of a candidate key

Partial Dependency: **LHS is a proper subset of CK and RHS is a non prime attribute**

- For 2nd NF, there should be no partial dependency

- Step 1. Find all candidate keys from functional dependencies of a relation
- Step 2. Find Prime and Non-Prime attributes
- Step 3. Check partial dependency for all the dependencies in a relation
  - If any partial dependency exists then that relation is not in 2nd normal form

THIRD NORMAL FORM

Third Normal Form

Conditions for third normal form (3rd NF):

- Relation should be in 2nd NF and
- And there should be no transitive dependency
  - Non prime attribute should not be determined by non prime attribute

Note: Simple condition for 3rd NF - LHS must be CK(or SK) or RHS must be PA

- Using this condition, you can directly check if relation is in 3rd NF or not.

BOYCE CODD NORMAL FORM

Boyce Codd Normal Form

- Condition for Boyce Codd Normal Form (BCNF):
  - Relation should be in 3rd NF and
  - And all attributes should be functional dependent on CK
- Example: FD: {rollno -> name, rollno -> voterid, voterid -> age, voterid -> rollno}; CK: {rollno, voterid}

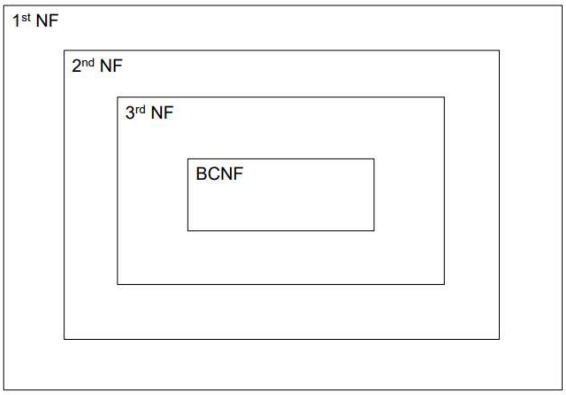
rollno	name	voterid	age
1	Ram	r0123	20
2	Shyam	v0546	21
3	Ram	r0678	22
4	Mohan	r0765	23

- Simple Condition: LHS of each FD should be CK or SK

Highly Normalized Database

- Advantage of highly normalized database
  - Less redundancy
- Disadvantage of highly normalized database
  - It has more number of tables, which takes more query access time

Expressive Power



- 1st NF has highest expressive power and BCNF has highest restriction

Decomposition

- Decomposition removes redundancy, anomalies and inconsistencies from a database by dividing the table into multiple tables
  - When we convert a relation to higher normal form, we decompose the relation
- Properties of Decomposition
  - Lossless Decomposition
  - Dependency Preserving Decomposition

A decomposition is a **lossless decomposition** if there is no loss of information by decomposing a relation (or replacing a relation with two relations)

Criteria for common attribute while decomposing a table

- Common attribute must be CK or SK of either R1 or R2 or both

Let R be a relation schema and let R_1 and R_2 form a decomposition of R

We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with the two relation schemas R_1 and R_2

Formally,

$$\prod_{R_1}(r) \bowtie \prod_{R_2}(r) = r$$

And, conversely a decomposition is lossy if

$$r \subset \prod_{R_1}(r) \bowtie \prod_{R_2}(r)$$

Simple conditions to check if decomposition is lossless

- $Attribute(R_1) \cup Attribute(R_2) = R$
- $Attribute(R_1) \cap Attribute(R_2) \neq \emptyset$
- Common attribute is super key (or candidate key) of either R1 or R2 or both

Equivalence of Functional Dependency

Equivalence of Functional Dependency

- F & D two functional dependencies are logical equivalent (F≡G) if
  - F covers G, or $G \subseteq F$: every FD's of G implied in F

    and
  - G covers F, or $F \subseteq G$: every FD's of F implied in G

Simple Method

- To check $G \subseteq F$
  - For each FD in G, pick it's LHS and take closure from F
  - If all closure from F are able to determine FD in G then $G \subseteq F$
- To check $F \subseteq G$
  - For each FD in F, pick it's LHS and take closure from G
  - If all closure from G are able to determine FD in F then $F \subseteq G$

Dependency Preserving Decomposition

- Example 1
  - $R = (A, B, C)$
    $F = \{A \rightarrow B, B \rightarrow C\}$
    Key = $\{A\}$. R is not in BCNF.
  - How to do decomposition and what will be attributes in decomposed relations?
    - First, the dependency that is creating a problem, make a separate relation (r2) for that. And attributes of that separate relation will be generated by taking closure of attribute present on the left side of problem creating dependency
    - Now, we make a new separate relation (r1) and attribute of this new relation will be remaining attributes that are not present in first relation (r2) and candidate key of first relation (r2)

Dependency Preserving Decomposition

- Method to find functional dependencies in decomposed relation (DDR) or $DDR \subseteq F$
  - Place all the possible dependencies that are satisfying original relation dependency, F
    - First, place maximum number of possible dependencies in decomposed relation
    - For each FD in decomposed relation, pick it's LHS and take closure from original relation FD, F
      - If closure from original relation is able to determine FD in decomposed relation then we keep this dependency otherwise, discord it
- Method to check if decomposed FDs cover original relation or $F \subseteq DDR$
  - For each FD in original relation, pick it's LHS and take closure from decomposed relation FD
    - If closure from decomposed relation is able to determine FD in original relation then we say that it is a dependency preserving decomposition

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
  - **Set of operations used to perform a logical unit of work**

Two main issues to deal with:

- Failures, such as hardware failures and system crashes
- Concurrent execution of multiple transactions

ACID PROPERTY

Atomicity requirement

- If the transaction fails after step 3 and before step 7, money will be "lost" leading to an inconsistent database state
  - Failure could be due to software or hardware
- The system should ensure that updates of a partially executed transaction are not reflected in the database
  - Either All Operations or No Operations

Durability requirement — Once the user has been notified that the transaction has completed (i.e., the transfer of the 1000 INR has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures. Permanent changes in database after a transaction is completed successfully

Consistency requirement in above example:

- The sum of A and B is unchanged by the execution of the transaction

In general, consistency requirements include

- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints
  - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database
- During transaction execution the database may be temporarily inconsistent
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Before a transaction starts and after the transaction completed, sum of the money should be same.

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B)	

- Isolation can be ensured by running transactions **serially(**one after other**)**

- Isolation can be ensured by running transactions **serially(**one after other**)**
  - Can we convert two parallel transactions in serial transaction conceptually.
    Why to convert: Serial Schedules are always **consistent**
- But, executing multiple transactions concurrently has significant benefits
  - Increased processor and disk utilization
  - Reduced average response time

and Information Systems 5.6

Transaction State

- **Active** – the initial state; a transaction stays in this state while it is executing
  - transaction is main memory and it is getting executed
- **Partially committed** – after the final statement has been executed
  - transaction has completed all the operations except commit
- **Failed** – after the discovery that normal execution can no longer proceed
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion
- **Terminated** – free the resources that were are used by transaction

Schedules

Schedule – a sequences of instructions that specify the **chronological order in which instructions of concurrent transactions are executed**

- A schedule for a set of transactions must consist of all instructions/operations of those transactions
- Must preserve the order in which the instructions appear in each individual transaction

A transaction that successfully completes its execution will have a commit instructions as the last statement

- By default transaction assumed to execute commit instruction as its last step

A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedules are two types: Serial Schedule and Parallel Schedule

Serial Schedule	Parallel Schedule
Transactions execute one after other	Transactions execute concurrently
Consistent	Can be Inconsistent
High waiting time	Less waiting time
Low throughput	High throughput
Low performance	High performance

Conflicts

Conflicts may occur when read or write operations are performed on **same data** by **different transactions**

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

Can lead to the undoing of a significant amount of work

The above schedule is cascading schedule

Problem due to cascading rollback

- Poor CPU utilization
- Less performance

Strict Recoverable Schedule

- T_i writes before T_j writes or reads, then T_j must read or write after T_i commits or aborts then only the schedule will be strict recoverable.
  - In other words, T_j can read or write updated or written value of T_i only after T_i commits/aborts.
  - Strict schedule is strict in nature

Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.

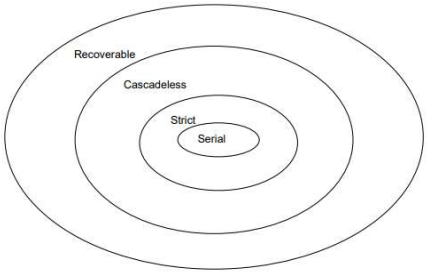
- **Transactions in a Schedule have ability to recover if any failure happens**

Cascadeless schedules — cascading rollbacks cannot occur:

- For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- No Write-Read problem

Every Cascadeless schedule is also recoverable

It is desirable to restrict the schedules to those that are cascadeless



Serializability

A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **Conflict serializable**
2. **View serializable**

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- If **I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule**.

Simplified view of transactions

We ignore operations other than **read** and **write** instructions

We assume that transactions may perform arbitrary computations on data in **local buffers** in between reads and writes.

Our simplified schedules consist of only **read** and **write** instructions.

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

Conflict Serializability (Cont.)

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

- For a schedule S, there can be many conflict equivalent schedules. If any conflict equivalent is a serial schedule, then given schedule is conflict serializable

Precedence graph — a direct graph where the vertices are the transactions (names)

We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.

- Check the conflict pair with other transaction in schedule and draw edge

A schedule is conflict serializable if and only if its precedence graph is acyclic.

- If no loop or cycle in precedence graph then schedule is conflict serializable

Serializability order can be obtained by a *topological sorting* of the graph.

- Sort based on indegree
  - Lowest indegree node will be first in sequence

View Serializability

Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,

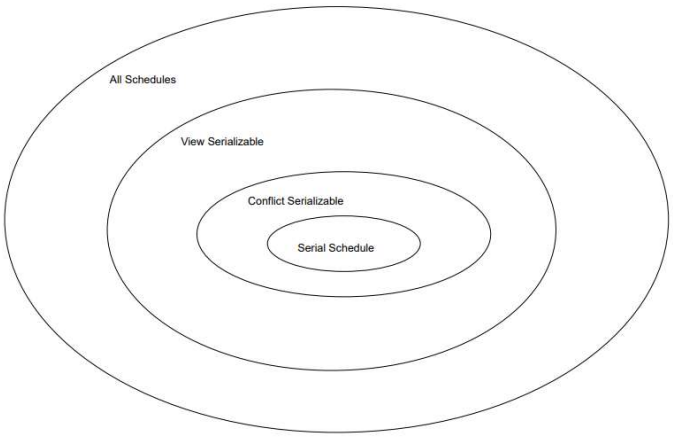
1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q (**Initial Read**).
2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j (**WR Sequence**).
3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S' **(Final Write)**.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

A schedule S is **view serializable** if it is view equivalent to a serial schedule.

Every conflict serializable schedule is also view serializable.



Concurrency Control

- A database must provide a **mechanism that will ensure that all possible schedules are**
  - either **conflict or view serializable**, and
  - are **recoverable and preferably cascadeless**

Concurrency Control Protocols

- Shared-Exclusive Locking
- Two Phase Locking (2PL)
- Basic Timestamp Ordering Protocol

Shared-Exclusive Locking

A lock is a mechanism to control concurrent access to a data item

Data items can be locked in two modes :

1. **exclusive** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction. **If a transaction locked the data item in exclusive mode, then that transaction is allowed to perform both read and write**
2. **shared** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction. **If a transaction locked the data item in shared mode, then that transaction is allowed to perform only read operation**

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix

Deadlock

- Consider the partial schedule

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

- Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back

can proceed only after request is granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.

Locking is *not sufficient* to guarantee serializability.

Shared-Exclusive Locking may suffer from irrecoverability

T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.

- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Starvation 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. Example: A only common room in family

The Two-Phase Locking Protocol

A protocol which ensures conflict-serializable schedules.

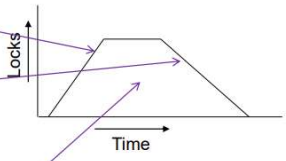
Phase 1: **Growing Phase**

- Transaction obtain locks
- Transaction do not release locks

Phase 2: **Shrinking Phase**

- Transaction release locks
- Transaction do not obtain locks

The protocol **assures serializability**. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its first unlock).



May not free from irrecoverabilty

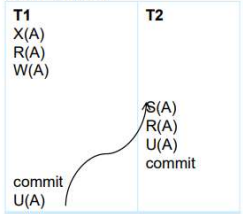
May not free from cascading rollback

May not free from deadlock

May not free from starvation

Strict 2PL, Rigorous 2PL, Conservative 2PL

- Strict 2PL
  - It should satisfy the basic 2PL and all exclusive locks should hold until commit/abort- Cascadeless, recoverable (strict)



- Rigorous 2PL
  - It should satisfy the basic 2PL and all shared, exclusive locks should hold until commit/abort
- Deadlock, starvation still can occur
- Conservative 2PL is free from deadlock

Timestamp-Based Protocols

- Each transaction T_i is issued a timestamp TS(T_i) when it enters the system.
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps strictly greater than earlier ones
    - Tells the order when a Transaction enters into system
  - Timestamp could be based on a logical counter
    - Real time may not be unique
- Timestamp-based protocols manage concurrent execution such that **time-stamp order = serializability order**
  - The transaction that comes earlier will execute earlier

Suppose a transaction T_i issues a **read**(Q)

1. If TS(T_i) < **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
   - Hence, the **read** operation is rejected, and T_i is rolled back.
2. If TS(T_i) ≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

Basic Timestamp Ordering Protocol

- Always ensure serializability
- Free from deadlock
- Do not ensure recoverable or cascadeless schedules

Strict Timestamp Ordering Protocol

- A Transaction T that issues a R_item(X) or W_item(X) **such that TS(T) > W_TS(X)** has its read or write operation delayed until the Transaction **T'** that wrote the values of X has committed or aborted
- Ensures recoverable and cascadeless schedules
- Free from deadlock
- Ensures serializability

The **timestamp ordering (TSO) protocol**

- Maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.
- Imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in timestamp order
  - Out of order operations cause transaction rollback

Suppose that transaction T_i issues **write**(Q).

1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
   - Hence, the **write** operation is rejected, and T_i is rolled back.
2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
   - Hence, this **write** operation is rejected, and T_i is rolled back.
3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Architecture of Computer System

Three main components of Computer System: CPU, Main memory (RAM), Secondary memory (Hard Disk)

Data is permanently stored in SM

When CPU execute a query, data is brought into PM then CPU perform the operation and changes are stored back into SM

- This is called I/O cost

Secondary Memory (SM) is divided into same size of blocks or pages

Primary memory (PM) is also divided into same size of blocks

Block size in SM = Block size in PM

Data (or records of a table) in SM can be ordered or unordered

- A block usually can contain multiple records

Why Indexing

- Indexing is used to reduce I/O cost
- It is used to speed up access to desired data
- Our aim in indexing is to transfer/call less no of blocks from SM to PM

Indexing mechanisms used to speed up access to desired data.

Search Key - attribute to set of attributes used to look up records in a file.

An **index file** consists of records (called **index entries**) of the form

search-key	pointer

Index files are typically much smaller than the original file

Index table block size same as block size of SM or PM

Index Table is also stored in SM permanently and once execution starts, the index table is first brought back to RAM and searched for desired key

Key in Index Table is always sorted and unique

- Consider a Hard Disk (HD) with Block Size 1000 bytes, each record size is 250 bytes, total records are 10000 and the data entered in HD without any order (unordered).
  - Find average time complexity (I/O cost) to search a record from Index Table where index table size is 20 Byte (10B for key + 10B for pointer)
- Solution
  - Index table block size = block size in HD = 1000B
  - No of entries in Index table = 1000B/20B = 50
  - If Sparse indexing, total no of entries in Index table = no of blocks in HD = 2500
  - No of blocks in Index table = 2500/50 = 50
  - Index is sorted so search time in index = log(50)
  - Total search time = index table search time + 1 = log(50)+1 = 7
  - If Dense Indexing, search time = log (10000/50) + 1 = 9

- Types of Indexing: Dense and Sparse
- **Dense index** — Index record appears for every search-key value in the file
  - Example: Figure that is shown in the last slide (slide 6.9) is an example of Dense Index
- **Sparse Index**: contains index records for only some search-key
  - Applicable when records are sequentially **ordered on search-key**
  - To locate a record with search-key value *K* we:
    - Find index record with largest search-key value < *K*
    - Search file sequentially starting at the record to which the index record points

Types of Indexes

- Primary Index
- Clustered Index
- Secondary Index

	Key	**Non Key**
Ordered File	Primary Index	Clustered Index
Unordered File	Secondary Index	Secondary Index

When we can apply Primary Index

- Data is **sorted**
- Search key is **unique or primary key**

Primary Index follows **sparse indexing**

Number of entries in Index Table = Number of blocks in HD

To locate a record with search-key value *K* we:

- Find index record with largest search-key value < *K*
- Search file sequentially starting at the record to which the index record points

Search Time in Primary Index = log(N) + 1; where N is the number of blocks in index table

Atmost one primary index for database table

- Clustered index is used when there is a **ordered** file and search key is a **non key**
- Clustered Index follows **sparse indexing**
- Best case search time = log(N) + 1; where N is the number of blocks in index table
- Worst case search time = log(N) + 1 + *1 (this can be more than 1, worst case no of blocks in HD*)
- Atmost one clustering index for database table

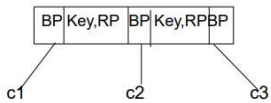
- Secondary indexed is used when data is **unordered**
- No of records in Index Table = No of records in HD
- Secondary indices have to be **dense**
- Search complexity = log(N) + 1, where N is number of blocks in Index table (where secondary search based on KEY)
- If secondary search is based on non key then we need to maintain intermediate layer that is block of record pointers
  - Search complexity = log(N) + 1 + 1

Dynamic Multilevel Index

- When Index size increases, we create Index table for original Index table
  - There can be index table for index table until last table is a single block
  - It creates problem while inserting or deleting a record in a table
    - Tree Data Structure is used to avoid the problem

Tree?. B-Tree is a balanced tree (leaf at same level). B-Tree properties:

- Block pointer or tree pointer
- Keys
- Data pointer or record pointer



- Order of a B-Tree (p): Max number of children a node can have

Children	Root	Intermediate or Leaf Node
Max	p	p
Min	2	$\lceil p/2 \rceil$

- Keys and Record pointer always have a 1 less than p (or order no of children in a node)
- Data is inserted in sorted order, like binary search tree

B-Tree vs. B⁺-Tree

B-Tree	B⁺ - Tree
Data is stored in leaf as well as internal nodes	Data is stored only in leaf nodes
Searching is slower	Searching is faster
No redundant search key present	Redundant keys would present
Leaf nodes are not linked together	Leaf nodes are linked together

Introduction to Structure Query Language

Structure Query Language (SQL): Domain specific and declarative language

In 1970, E.F. Codd develops relational database concepts: published a research paper while working at IBM San Jose Research Laboratory

During 1974-1979, Sequel (Structured English Query Language) was created at IBM, renamed to SQL Later

In 1979, Oracle markets first DB with SQL

In 1986, ANSI SQL standards were released and updated later wards (1989, 1992, 1999, 2003, …)

Most DBMS are SQL-99 compliant, with partial SQL-2003 compliant

Now Database major players: Oracle, IBM, Microsoft, MySQL

Sub Language

- Data Definition Language
  - Used to define structure of a table
- Data Manipulation Language
  - Used to manipulate database records
- Data Query Language
  - Used to access required data from database tables
- Data Control Language
  - Used for transaction based operations and security

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation (or table)
- The type of values associated with each attribute
- The Integrity constraints
- Examples
  - Create Table
  - Drop Table
    - Delete table
  - Alter Table
    - Add and remove columns in a table
  - Truncate
    - Delete all the data inside a table
  - Integrity constraints: primary key, foreign key, alternate key
  - Rename

ID	Roll No	Department

Table: Student

char(n). Fixed length character string, with user-specified length *n*.

varchar(n). Variable length character strings, with user-specified maximum length *n*.

int. Integer (a finite subset of the integers that is machine-dependent).

numeric(p,d). Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. (ex., **numeric**(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)

Data Manipulation Language

- Data Manipulation Language (DML) is used to manipulate the data records
- **Insert**
  - **insert into** *Student* **values** ('4', 'Ram', 'CS');
- **Delete**
  - Remove all tuples from the *student* relation
    - **delete from** *Student*
  - Remove student with ID 2
    - **delete from** *Student* where *ID = 2;*
- **Update**
  - Update a tuple from the *student* relation
    - update Student set Department = 'CS' where ID = 3;

Data Query Language

Data Query Language (DQL) is used to access required data from database tables

Other DQL commands

- Group By
- Having

Data Control Language

- Data Control Language (DCL) is used for transaction based operations and security
- Operations
  - Grant
    - Give privileges to a user over table
  - Revoke
    - Remove privileges from user over table
  - Rollback
    - If transaction is failed, rollback it
  - Commit
    - Transactions completed successfully, save in database
  - Save Point
    - Save some part of execution in DB

Alter	**Update**
DDL	DML
Make changes in relation or table structure	Make changes in data
alter table employee add address varchar (10);	update employee set salary = salary*2 where ID =1;

Delete	Drop	Truncate
DML Command	DDL Command	DDL Command
Delete all rows. delete from Student;	Delete table structure. drop table Student;	Delete all rows. truncate table Student;
Can give condition. delete from Student where ID = 1;	No condition	No condition
Can rollback before commit. Use logs	No Rollback	No rollback
Slower	Faster	Faster

```
insert into instructor
    select ID, name, dept_name, 18000
    from   student
    where  dept_name = 'Music' and total_cred > 144;
```

- Select Ename from Emp where Eid in (Select Distinct Eid from Project);

Write a SQL query to find employee name who is taking second highest salary

- Select E_name from Emp where Salary = (Select Max(Salary) from Emp where Salary <> (Select Max(Salary) from Emp));

Select Dept, count(Dept) from Emp Group By(Dept);

Select Dept from Emp Group By(Dept) having count(*) <2;

Now employee name also can be found using nested query

- Select E_name from Emp where Dept in (Select Dept from Emp Group By(Dept) having count(*) <2);

Find all employees detail who work in a department or false

- Select * from Emp where exists (Select * from Dept where Emp.eid= Dept.eid)

Find N-th highest salary

- Will use correlated nested query, it processes top to bottom
- Select ID, Salary from Emp e1 where N-1 = (Select count (distinct Salary) from Emp e2 where e2.Salary > e1.Salary)

Nested SubQuery	Correlated SubQuery (or Correlated SubQuery)	Joins
Bottom Up Approach	Top Down Approach	Cross Product + Condition
Select * from Emp where eid in (Select eid from Dept);	Select * from Emp where exits (Select eid from Dept where Emp.eid = Dept.eid);	Select Emp.eid, Emp.name from Emp , Dept where Emp.eid = Dept.eid;