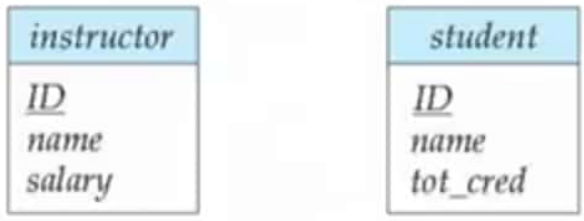
Entity Sets

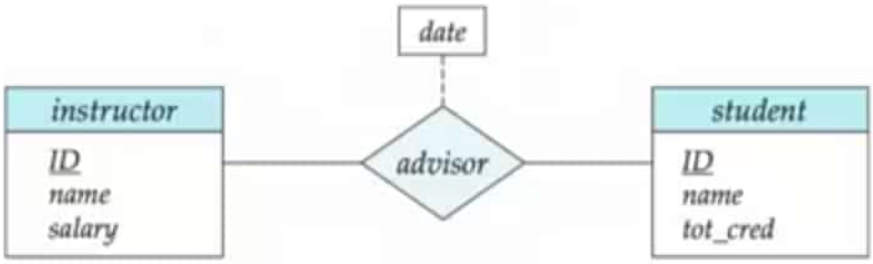
- Entities can be represented graphically as follows:
  - Rectangles represent entity sets.
  - Attributes listed inside entity rectangle
  - Underline indicates primary key attributes



Relationship Sets

- Diamonds represent relationship sets.

Relationship Sets with Attributes



Expressing Weak Entity Sets

- In E-R diagrams, a weak entity set is depicted via a double rectangle
- We underline the discriminator of a weak entity set with a dashed line
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond
- Primary key for *section* – (*course_id, sec_id, semester, year*)

PPD

First Normal Form (1NF)

- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in **first normal form** if
  - the domains of all attributes of R are atomic
  - the value of each attribute contains only a single value from that domain

First Normal Form (Cont'd)

PPD

- Better to have 2 relations:

Customer Name

Customer ID	First Name	Surname
123	Pooja	Singh
456	San	Zhang
789	John	Doe

Customer Telephone Number

Customer ID	Telephone Number
123	555-861-2025
123	192-122-1111
456	(555) 403-1659 Ext. 53
456	182-929-2929
789	555-808-9633

- One-to-Many relationship between parent and child relations
- Incidentally, sattisfies 2NF and 3NF

Source: https://en.wikipedia.org/wiki/First_normal_form

Closure of Functional Dependencies (Cont.)

- Additional rules:
  - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (**union**)
  - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (**decomposition**)
  - If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds (**pseudotransitivity**)

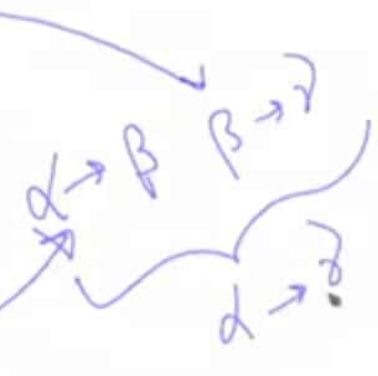
  The above rules can be inferred from Armstrong's axioms

Closure of Attribute Sets

- Given a set of attributes α, define the ***closure*** of α **under** F (denoted by α^+) as the set of attributes that are functionally determined by α under F

- Algorithm to compute α^+, the closure of α under F

```
result := α;
while (changes to result) do
    for each β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ
        end
```



Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$

$AG \rightarrow ABCGHI$

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if α is a superkey, we compute α^+, and check if α^+ contains all attributes of R.
- Testing functional dependencies
  - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.
  - That is, we compute α^+ by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful
- Computing closure of F
  - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g.: on RHS: $\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow CD\}$ can be simplified to $\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow D\}$
      - *In the forward:* (1) $A \rightarrow CD \rightarrow A \rightarrow C$ and $A \rightarrow D$ (2) $A \rightarrow B,\ B \rightarrow C \rightarrow A \rightarrow C$
      - *In the reverse:* (1) $A \rightarrow B,\ B \rightarrow C \rightarrow A \rightarrow C$ (2) $A \rightarrow C,\ A \rightarrow D \rightarrow A \rightarrow CD$
    - E.g.: on LHS: $\{A \rightarrow B,\ B \rightarrow C,\ AC \rightarrow D\}$ can be simplified to $\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow D\}$
      - *In the forward:* (1) $A \rightarrow B,\ B \rightarrow C \rightarrow A \rightarrow C \rightarrow A \rightarrow AC$ (2) $A \rightarrow AC,\ AC \rightarrow D \rightarrow A \rightarrow D$
      - *In the reverse:* $A \rightarrow D \rightarrow AC \rightarrow D$
- Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

Extraneous Attributes

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
  - Attribute A is **extraneous** in α if $A \in \alpha$
    and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - Attribute A is **extraneous** in β if $A \in \beta$
    and the set of functional dependencies
    $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.
- *Note:* Implication in the opposite direction is trivial in each of the cases above, since a "stronger" functional dependency always implies a weaker one
- Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
  - B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (I.e. the result of dropping B from $AB \rightarrow C$).
  - A+ = AC in $\{A \rightarrow C, AB \rightarrow C\}$

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

To Identify whether a decomposition is lossy or lossless, it must satisfy the following conditions :

- $R_1 \cup R_2 = R$
- $R_1 \cap R_2 \neq \Phi$ *and*
- $R_1 \cap R_2 \rightarrow R_1$ *or* $R_1 \cap R_2 \rightarrow R_2$

PPD

First Normal Form (1 NF)

- A relation is in first Normal Form if and only if all underlying domains contain atomic values only
- In other words, a relation doesn't have multivalued attributes (MVA)
- Example:
  - ***STUDENT(Sid, Sname, Cname)***

Students

SID	Sname	Cname
S1	A	C,C++
S2	B	C++, DB
S3	A	DB

SID : Primary Key

MVA exists → Not in 1NF

Students

SID	Sname	Cname
S1	A	C
S1	A	C++
S2	B	C++
S2	B	DB
S3	A	DB

SID : Primary Key

No MVA → In 1NF

Source: http://www.edugrabs.com/normal-forms/#fnf

PPD

Second Normal Form (2 NF)

- Relation ***R*** is in Second Normal Form (2NF) only iff :
  - ***R*** should be in 1NF and
  - ***R*** should not contain any *Partial Dependency*

Partial Dependency:

Let ***R*** be a relational Schema and ***X,Y,A*** be the attribute sets over ***R*** where
X: Any Candidate Key, ***Y:*** Proper Subset of Candidate Key, and ***A:*** Non Key Attribute

If Y → A exists in R, then R is not in 2 NF.

(Y → A) is a Partial dependency only if
- ***Y:*** Proper subset of Candidate Key
- ***A:*** Non Prime Attribute

Source: http://www.edugrabs.com/2nf-second-normal-form

PPD

Second Normal Form (2 NF)

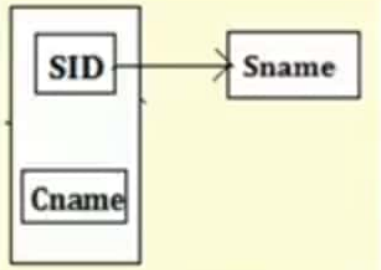
- Example:
  - ***STUDENT(Sid, Sname, Cname)*** (already in 1NF)

Students:

SID	Sname	Cname
S1	A	C
S1	A	C++
S2	B	C++
S2	B	DB
S3	A	DB

(SID, Cname): Primary Key

- **Redundancy?**
  - *Sname*
- **Anomaly?**
  - *Yes*



Functional Dependencies:
{SID,Cname} → Sname
SID → Sname

Partial Dependencies:
SID → Sname (as SID is a Proper Subset of Candidate Key {SID,Cname})

Post Normalization

R1:

SID	Sname
S1	A
S2	B
S3	A

{SID}: Primary Key

R2:

SID	Cname
S1	C
S1	C++
S2	C++
S2	DB
S3	DB

{SID,Cname}: Primary Key

The above two relations R1 and R2 are
1. Lossless Join
2. 2NF
3. Dependency Preserving

PPD

Third Normal Form (3 NF)

Let ***R*** be the relational schema.

- [E. F. Codd ,1971] ***R*** is in 3NF only if:
  - ***R*** should be in 2NF
  - ***R*** should not contain *transitive dependencies (OR,* Every non-prime attribute of R is non-transitively dependent on every key of R)
- [Carlo Zaniolo, 1982] Alternately, ***R*** is in 3NF iff for each of its functional dependencies $X \rightarrow A$, at least one of the following conditions holds:
  - X contains A (that is, A is a subset of X, meaning $X \rightarrow A$ is trivial functional dependency), or
  - X is a superkey, or
  - Every element of A-X, the set difference between A and X, is a *prime attribute* (i.e., each attribute in $A - X$ is contained in some candidate key)
- [Simple Statement} A relational schema **R** is in 3NF if for every FD $\mathbf{X} \rightarrow \mathbf{A}$ associated with R either
  - $\mathbf{A} \subseteq \mathbf{X}$ (i.e., the FD is trivial) or
  - **X** is a superkey of **R** or
  - **A** is part of some key (not just superkey!)

Source: http://www.edugrabs.com/3nf-third-normal-form/

PPD

3NF Decomposition Algorithm

- Given: relation R, set F of functional dependencies
- Find: decomposition of R into a set of 3NF relation Ri
- Algorithm:
  1. Eliminate redundant FDs, resulting in a canonical cover Fc of F
  2. Create a relation Ri = XY for each FD $X \rightarrow Y$ in Fc
  3. If the key K of R does not occur in any relation Ri, create one more relation Ri=K

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - the decomposition is lossless
  - it may not be possible to preserve dependencies.

S#	3NF	BCNF
1.	It concentrates on Primary Key	It concentrates on Candidate Key.
2.	Redundancy is high as compared to BCNF	0% redundancy
3.	It may preserve all the dependencies	It may not preserve the dependencies.
4.	A dependency X → Y is allowed in 3NF if X is a super key or Y is a part of some key.	A dependency X → Y is allowed if X is a super key

Theory of MVDs

PPD

	Name		Rule
C-	**Complementation**	:	If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow \{R - (X \cup Y)\}$.
A-	**Augmentation**	:	If $X \twoheadrightarrow Y$ and $W \supseteq Z$, then $WX \twoheadrightarrow YZ$.
T-	**Transitivity**	:	If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow (Z - Y)$.
	Replication	:	If $X \rightarrow Y$, then $X \twoheadrightarrow Y$ but the reverse is not true.
	Coalescence	:	If $X \twoheadrightarrow Y$ and there is a W such that $W \cap Y$ is empty, $W \rightarrow Z$, and $Y \supseteq Z$, then $X \rightarrow Z$.

- A MVD $X \twoheadrightarrow Y$ in R is called a trivial MVD is
  - Y is a subset of X ($X \supseteq Y$) or
  - $X \cup Y = R$. Otherwise, it is a non trivial MVD and we have to repeat values redundantly in the tuples.

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Source: http://www.edugrabs.com/multivalued-dependency-mvd/

PPD

Relational Schema

- **books**(title, author_fname, author_lname, publisher, year, ISBN_no, accession_no)
- **book_issue**(members, accession_no, doi)
- **members**(member_no, member_type)
- **quota**(member_type, max_books, max_duration)
- **students**(member_no, student_fname, student_lname, roll_no, department, gender, mobile_no, dob, degree)
- **faculty**(member_no, faculty_fname, faculty_lname, id, department, gender, mobile_no, doj)
- **staff**(staff_fname, staff_lname, id, gender, mobile_no, doj)

PPD

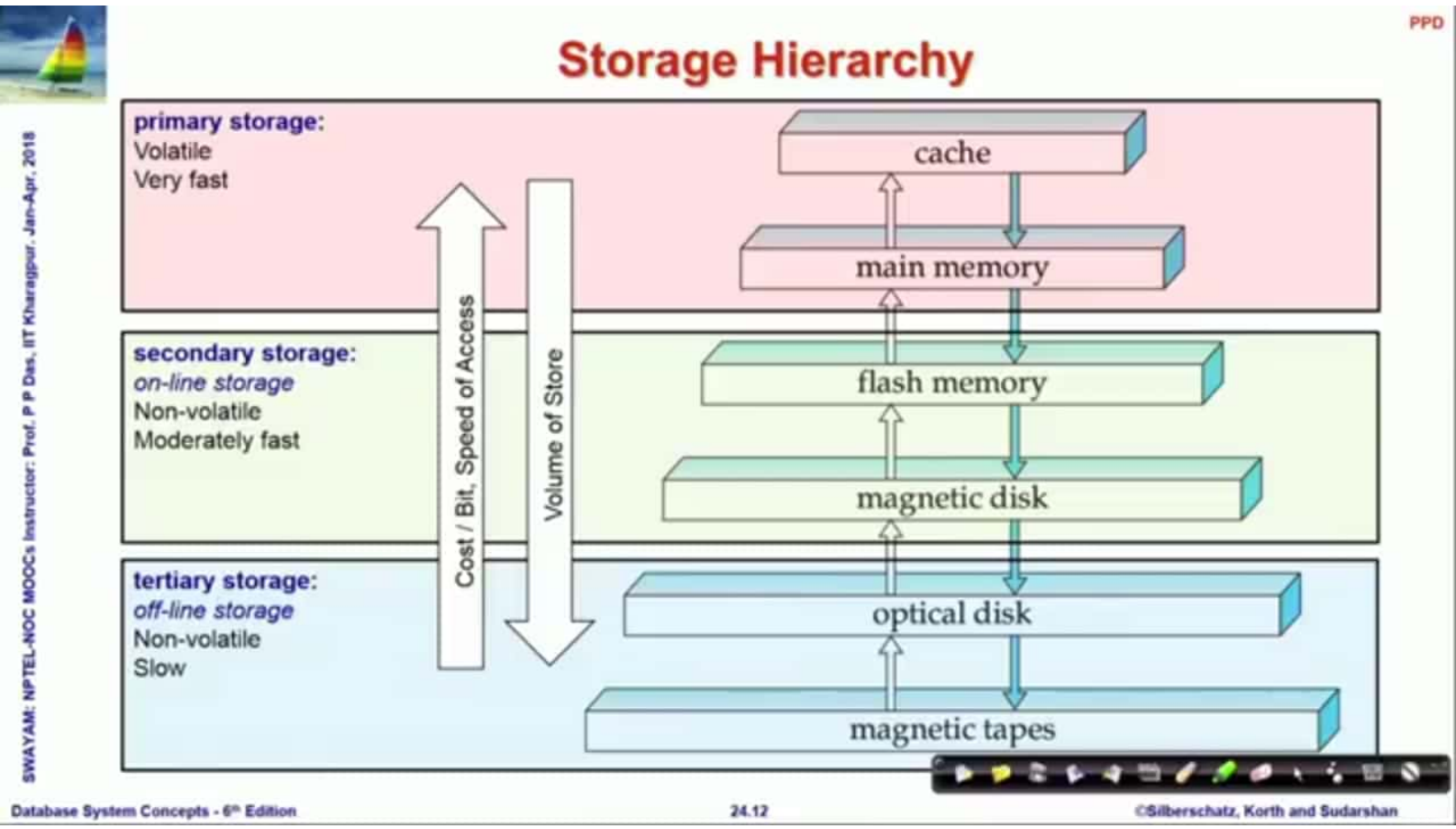
Schema Refinement

- **books**(title, author_fname, author_lname, publisher, year, ISBN_no, accession_no)
  - ISBN_no → title, author_fname, author_lname, publisher, year
  - accession_no → ISBN_no
  - Key: accession_no
- Redundancy of book information across copies
- Good to normalize:
  - **book_catalogue**(title, author_fname, author_lname, publisher, year, ISBN_no)
    - ISBN_no → title, author_fname, author_lname, publisher, year
    - Key: ISBN_no
  - **book_copies**(ISBN_no, accession_no)
    - accession_no → ISBN_no
    - Key: accession_no
- Both in BCNF. Decomposition is lossless join and dependency preserving

PPD

Schema Refinement – Final

- **book_catalogue**(title, author_fname, author_lname, publisher, year, ISBN_no)
- **book_copies**(ISBN_no, accession_no)
- **book_issue**(member_no, accession_no, doi)
- **quota**(member_type, max_books, max_duration)
- **members**(member_no, member_class, member_type, roll_no, id)
- **students**(student_fname, student_lname, roll_no, department, gender, mobile_no, dob, degree)
- **faculty**(faculty_fname, faculty_lname, id, department, gender, mobile_no, doj)
- **staff**(staff_fname, staff_lname, id, gender, mobile_no, doj)



Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. It consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track
    - Average seek time is 1/2 the worst case seek time
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head
    - Average latency is 1/2 of the worst case latency.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 rpm)

PPD

Search Data Structures

- Worst case time (n data items in the data structure):

Data Structure	Search	Insert	Delete	Remarks
Unordered Array	O(n)	O(1)	O(1)	The time to Insert / Delete an item is the time after the location of the item has been ascertained by Search.
Ordered Array	O(log n)	O(n)	O(n)	
Unordered List	O(n)	O(1)	O(1)	
Ordered List	O(n)	O(1)	O(1)	
Binary Search Tree	O(h)	O(1)	O(1)	

- Between an array and a list, there is a trade-off between search and insert/delete complexity
- For a BST of n nodes, log n <= h <= n, where h is the height of the tree
- A BST is balanced if h ~ O(log n) – this what we desire

PPD

Transaction Control Language (TCL)

- The following commands are used to control transactions.
  - **COMMIT** − to save the changes
  - **ROLLBACK** − to roll back the changes
  - **SAVEPOINT** − creates points within the groups of transactions in which to ROLLBACK
  - **SET TRANSACTION** − Places a name on a transaction

Source: http://www.tutorialspoint.com/sql/sql-transactions

PPD

TCL: COMMIT Command

- The COMMIT is the transactional command used to save changes invoked by a transaction to the database
- The COMMIT saves all the transactions to the database since the last COMMIT or ROLLBACK command
- The syntax for the COMMIT command is as follows:
  - SQL> DELETE FROM Customers WHERE AGE = 25;
  - SQL> COMMIT;

SQL> SELECT * FROM Customers;

Before DELETE

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

SQL> SELECT * FROM Customers;

After DELETE

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
3	kaushik	23	Kota	2000
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

Source: http://www.tutorialspoint.com/sql/sql-transactions

PPD

Transaction Control Language (TCL)

- The following commands are used to control transactions.
  - **COMMIT** − to save the changes
  - **ROLLBACK** − to roll back the changes
  - **SAVEPOINT** − creates points within the groups of transactions in which to ROLLBACK
  - **SET TRANSACTION** − Places a name on a transaction
- Transactional control commands are only used with the **DML Commands** such as
  - INSERT, UPDATE and DELETE only
  - They cannot be used while creating tables or dropping them because these operations are automatically committed in the database

Source: http://www.tutorialspoint.com/sql/sql-transactions

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.
  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 .
  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'

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
- Below is a schedule which is view-serializable but *not* conflict serializable

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

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
- Below is a schedule which is view-serializable but *not* conflict serializable

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
  - T_{27}–T_{28}–T_{29}
  - The one read(Q) instruction reads the initial value of Q in both schedules and
  - T_{29} performs the final write of Q in both schedules
- T_{28} and T_{29} perform write(Q) operations called **blind writes**, without having performed a read(Q) operation

View Serializability: Example 1

- Check whether the schedule is view serializable or not?
  - S : R2(B); R2(A); R1(A); R3(A); W1(B); W2(B); W3(B);
- Solution:
  - With 3 transactions, total number of schedules possible = 3! = 6
    - <T1 T2 T3>
    - <T1 T3 T2>
    - <T2 T3 T1>
    - <T2 T1 T3>
    - <T3 T1 T2>
    - <T3 T2 T1>
  - Final update on data items :
    - A : -
    - B : T1 T2 T3
    - Since the final update on B is made by T3, so the transaction T3 must execute after transactions T1 and T2.
    - Therefore, (T1,T2) → T3. Now, Removing those schedules in which T3 is not executing at last:
      - <T1 T2 T3>
      - <T2 T1 T3>

Source: http://www.edugrabs.com/how-to-check-for-view-serializabl

View Serializability: Example 1

PPD

- Check whether the schedule is view serializable or not?
  - S : R2(B); R2(A); R1(A); R3(A); W1(B); W2(B); W3(B);
- Solution:
  - Initial Read + Which transaction updates after read?
    - A : T2 T1 T3 (initial read)
    - B : T2 (initial read); T1 (update after read)
    - The transaction T2 reads B initially which is updated by T1. So T2 must execute before T1.
    - Hence, T2 → T1. Removing those schedules in which T2 is executing before T1:
    - <T2 T1 T3>
  - Write Read Sequence (WR)
    - No need to check here
  - Hence, view equivalent serial schedule is:
    - **T2 → T1 → T3**

T1 T2 T3
T2 T1 T3

Source: http://www.edugrabs.com/how-to-check-for-view-serializabl

Lock-Based Protocols: Example

- Given, T3 and T4, consider Schedule 2 (partial)
- Since *T3* is holding an exclusive mode lock on *B* and *T4* is requesting a shared-mode lock on *B*, *T4* is waiting for *T3* to unlock *B*
- Similarly, since *T4* is holding a shared-mode lock on *A* and *T3* is requesting an exclusive-mode lock on *A*, *T3* is waiting for *T4* to unlock *A*
- Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution
- This situation is called **deadlock**

T3:

```
lock-X(B);
read(B);
B := B − 50;
write(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(B);
unlock(A)
```

T4:

```
lock-S(A);
read(A);
lock-S(B);
read(B);
display(A + B);
unlock(A);
unlock(B)
```

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)	

Schedule 2

Lock-Based Protocols

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states
- On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur
- Deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states
- Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks
- Locking protocols restrict the set of possible schedules
- The set of all such schedules is a proper subset of all possible serializable schedules
- We present locking protocols that allow only conflict-serializable schedules, and thereby ensure isolation

The Two-Phase Locking Protocol

- This protocol ensures conflict-serializable schedules
- Phase 1: Growing Phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking Phase
  - Transaction may release locks
  - Transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points**

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration)
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits
- But the schedule may not be cascade-free, and may not even be recoverable

Timestamp-Based Protocols

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order
- 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))

Formal Relational Query Language

- Relational Algebra
  - Procedural and Algebra based
- Tuple Relational Calculus
  - Non-Procedural and Predicate Calculus based
- Domain Relational Calculus
  - Non-Procedural and Predicate Calculus based

PPD

Equivalence of RA, TRC and DRC

Select Operation

R = (A, B)

Relational Algebra: $\sigma_{B=17}(r)$

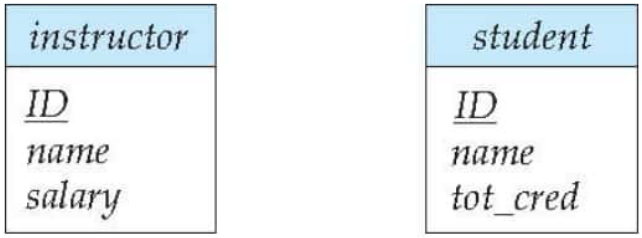
Tuple Calculus: $\{t \mid t \in r \wedge B = 17\}$

Domain Calculus: $\{\langle a, b\rangle \mid \langle a, b\rangle \in r \wedge b = 17\}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf

Entity Sets

- Entities can be represented graphically as follows:
  - Rectangles represent entity sets.
  - Attributes listed inside entity rectangle
  - Underline indicates primary key attributes



Relationship Sets

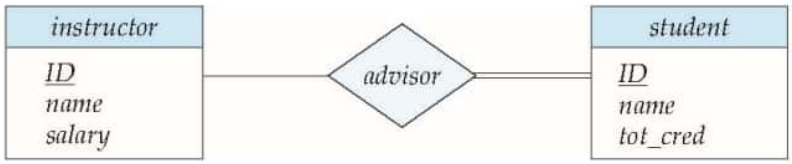
- Diamonds represent relationship sets.

Cardinality Constraints

- We express cardinality constraints by drawing either a directed line (→), signifying "one," or an undirected line (—), signifying "many," between the relationship set and the entity set.
- One-to-one relationship between an *instructor* and a *student* :
  - A student is associated with at most one *instructor* via the relationship *advisor*
  - A *student* is associated with at most one *department* via *stud_dept*

Total and Partial Participation

- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set



  - participation of *student* in *advisor* relation is total
    - every *student* must have an associated instructor
- Partial participation: some entities may not participate in any relationship in the relationship set
  - Example: participation of *instructor* in *advisor* is partial

Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form *l..h*, where *l* is the minimum and *h* the maximum cardinality
  - A minimum value of 1 indicates total participation.
  - A maximum value of 1 indicates that the entity participates in at most one relationship
  - A maximum value of * indicates no limit.

Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors

Expressing Weak Entity Sets

- In E-R diagrams, a weak entity set is depicted via a double rectangle
- We underline the discriminator of a weak entity set with a dashed line
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond
- Primary key for *section* – (*course_id, sec_id, semester, year*)

Representing Entity Sets

- A strong entity set reduces to a schema with the same attributes

  student(ID, name, tot_cred)

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set

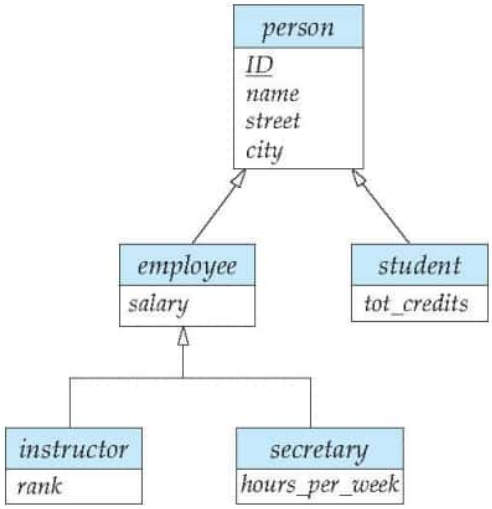
  section (course_id, sec_id, sem, year)

Specialization

- Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* "is a" *person*)
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked

Specialization Example

- **Overlapping** – *employee* and *student*
- **Disjoint** – *instructor* and *secretary*
- Total and partial



Representing Specialization via Schemas

- Method 1:
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

  - Drawback: Getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema

Representing Specialization as Schemas (Cont.)

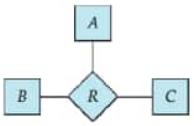
- Method 2:
  - Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary

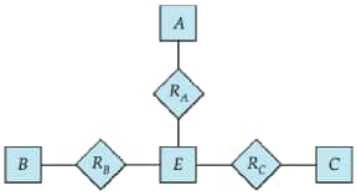
  - Drawback: *name*, *street* and *city* may be stored redundantly for people who are both students and employees

Converting Non-Binary Relationships to Binary Form

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
  - Replace R between entity sets A, B and C by an entity set E, and three relationship sets:
    1. R_A, relating E and A
    2. R_B, relating E and B
    3. R_C, relating E and C
  - Create an identifying attribute for E *and* add any attributes of R to E
  - For each relationship (a_i , b_i , c_i) in R, create
    1. a new entity e_i in the entity set E
    2. add (e_i , a_i) to R_A
    3. add (e_i , b_i) to R_B
    4. add (e_i , c_i) to R_C



(a)



(b)

Application Architectures

- Application layers
  - Presentation or user interface
    - **model-view-controller (MVC)** architecture
      - **model**: business logic
      - **view**: presentation of data, depends on display device
      - **controller**: receives events, executes actions, and returns a view to the user
  - **business-logic** layer
    - provides high level view of data and actions on data
      - often using an object data model
    - hides details of data storage schema
  - **data access** layer
    - interfaces between business logic layer and the underlying database
    - provides mapping from object model of business layer to relational model of database

Business Logic Layer

- Provides abstractions of entities
  - e.g. students, instructors, courses, etc
- Enforces **business rules** for carrying out actions
  - E.g. student can enroll in a class only if she has completed prerequsites, and has paid her tuition fees
- Supports **workflows** which define how a task involving multiple participants is to be carried out
  - E.g. how to process application by a student applying to a university
  - Sequence of steps to carry out task
  - Error handling
    - e.g. what to do if recommendation letters not received on time

- **book_copies**(ISBN_no, accession_no)
  - accession_no → ISBN_no
  - Key: accession_no
- Both in BCNF. Decomposition is lossless join and dependency preserving

Schema Refinement

- **book_issue**(member_no, accession_no, doi)

Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
  - data loss on power failure or system crash
  - physical failure of the storage device
- Can differentiate storage into:
  - **volatile storage:** loses contents when power is switched off
  - **non-volatile storage**:
    - Contents persist even when power is switched off
    - Includes secondary and tertiary storage, as well as batter-backed up main-memory

Physical Storage Media

- **Cache**
  - fastest and most costly form of storage
  - volatile
  - managed by the computer system hardware
- **Main memory**
  - fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
  - generally too small (or too expensive) to store the entire database
    - capacities of up to a few Gigabytes widely used currently
    - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
  - **Volatile**
    - contents of main memory are usually lost if a power failure or system crash occurs

Physical Storage Media (Cont.)

- **Flash memory**
  - Data survives power failure
  - Data can be written at a location only once, but location can be erased and written to again
    - Can support only a limited number (10K – 1M) of write/erase cycles
    - Erasing of memory has to be done to an entire bank of memory
  - Reads are roughly as fast as main memory
  - But writes are slow (few microseconds), erase is slower
  - Widely used in embedded devices such as digital cameras, phones, and USB keys

Physical Storage Media (Cont.)

- **Magnetic-disk**
  - Data is stored on spinning disk, and read/written magnetically
  - Primary medium for the long-term storage of data
    - typically stores entire database
  - Data must be moved from disk to main memory for access, and written back for storage
    - Much slower access than main memory
  - **direct-access**
    - possible to read data on disk in any order, unlike magnetic tape
  - Capacities range up to roughly 16~32 TB
    - Much larger capacity and cost/byte than main memory/flash memory
    - Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
  - Survives power failures and system crashes
    - disk failure can destroy data, but is rare

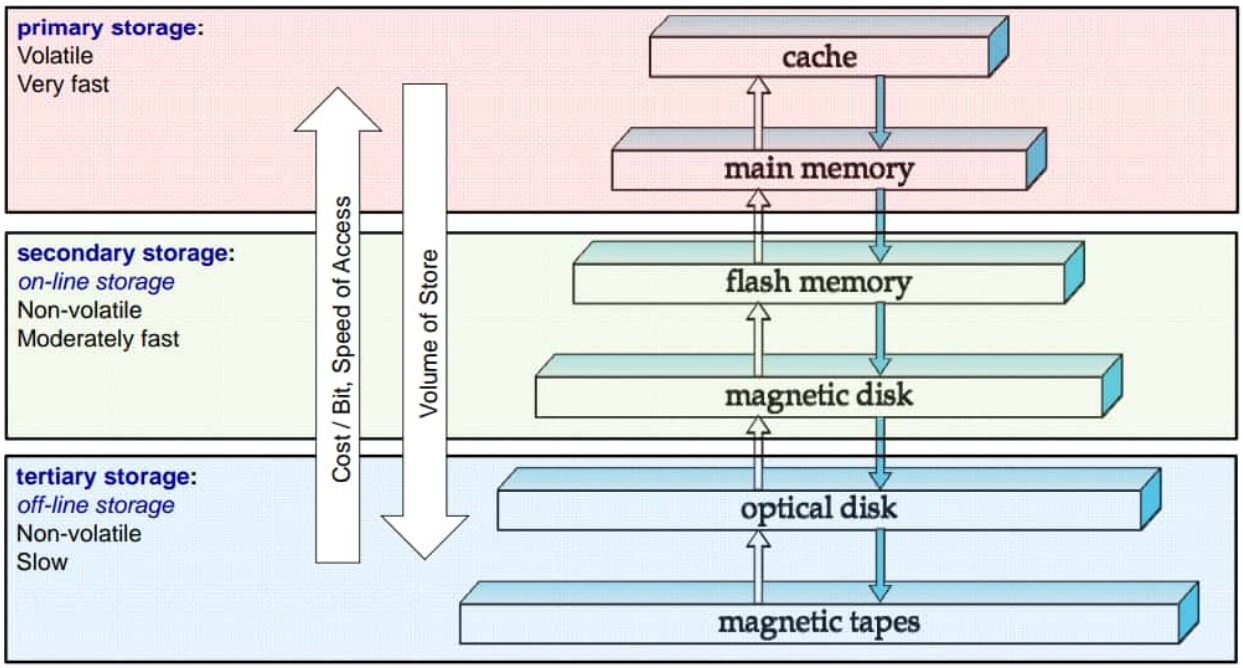
Physical Storage Media (Cont.)

- **Optical storage**
  - non-volatile, data is read optically from a spinning disk using a laser
  - CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
  - Blu-ray disks: 27 GB to 54 GB
  - Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
  - Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
  - Reads and writes are slower than with magnetic disk
  - **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

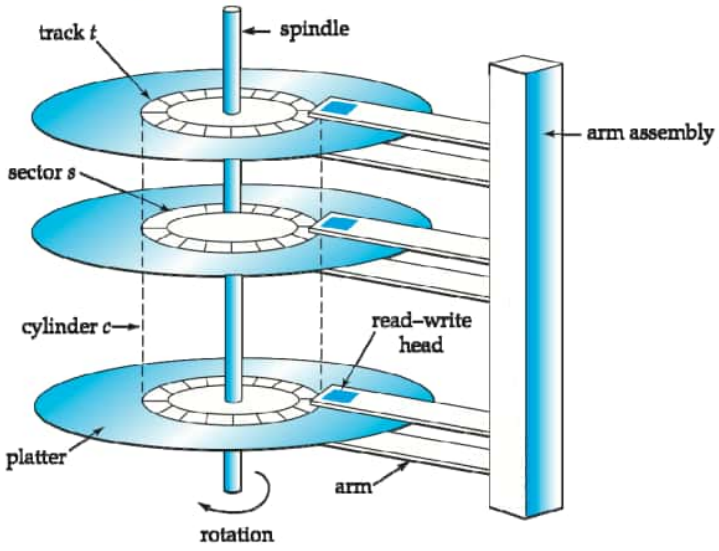
- **Tape storage**
  - non-volatile, used primarily for backup (to recover from disk failure), and for archival data
  - **sequential-access**
    - much slower than disk
  - very high capacity (40 to 300 TB tapes available)
  - tape can be removed from drive ⇒ storage costs much cheaper than disk, but drives are expensive
  - Tape jukeboxes available for storing massive amounts of data
    - hundreds of terabytes (1 terabyte = 10^{12} bytes) to even multiple **petabytes** (1 petabyte = 10^{15} bytes)

PPD

Storage Hierarchy



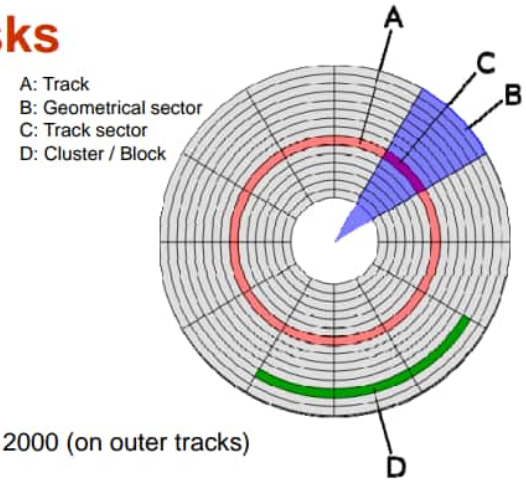
Magnetic Hard Disk Mechanism



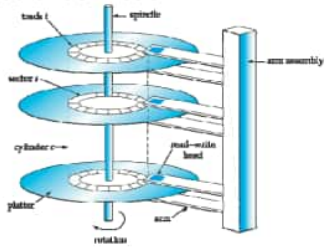
NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

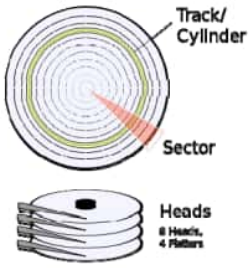
PPD

Magnetic Disks



- **Read-write head**
  - Positioned very close to the platter surface (almost touching it)
  - Reads or writes magnetically encoded information
- Surface of platter divided into circular **tracks**
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
  - multiple disk platters on a single spindle (1 to 5 usually)
  - one head per platter, mounted on a common arm.
- **Cylinder** *i* consists of *i*th track of all the platters



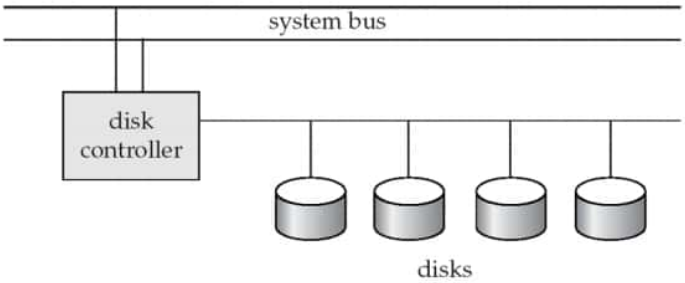


SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Magnetic Disks (Cont.)

- Earlier generation disks were susceptible to head-crashes
  - Surface of earlier generation disks had metal-oxide coatings which would disintegrate on head crash and damage all data on disk
  - Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted
- **Disk controller** – interfaces between the computer system and the disk drive hardware
  - accepts high-level commands to read or write a sector
  - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
  - Computes and attaches **checksums** to each sector to verify that data is read back correctly
    - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
  - Ensures successful writing by reading back sector after writing it
  - Performs remapping of bad sectors

Disk Subsystem



- Multiple disks connected to a computer system through a controller
  - Controllers functionality (checksum, bad sector remapping) often carried out by individual disks; reduces load on controller
- Disk interface standards families
  - ATA (AT adaptor) range of standards
  - SATA (Serial ATA)
  - SCSI (Small Computer System Interconnect) range of standards
  - SAS (Serial Attached SCSI)
  - Several variants of each standard (different speeds and capabilities)

Disk Subsystem

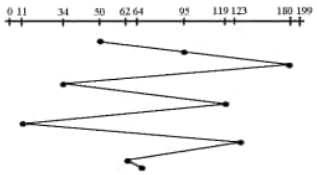
- Disks usually connected directly to computer system
- In **Storage Area Networks (SAN)**, a large number of disks are connected by a high-speed network to a number of servers
- In **Network Attached Storage (NAS)** networked storage provides a file system interface using networked file system protocol, instead of providing a disk system interface

Performance Measures of Disks

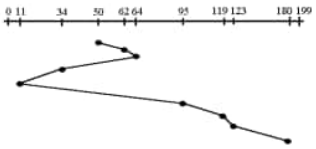
- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. It consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track
    - Average seek time is 1/2 the worst case seek time
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head
    - Average latency is 1/2 of the worst case latency.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 rpm)
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 100 MB per second max rate, lower for inner tracks
  - Multiple disks may share a controller, so rate that controller can handle is also important
    - E.g. SATA: 150 MB/sec, SATA-II 3Gb (300 MB/sec)
    - Ultra 320 SCSI: 320 MB/s, SAS (3 to 6 Gb/sec)
    - Fiber Channel (FC2Gb or 4Gb): 256 to 512 MB/s

Optimization of Disk-Block Access (Cont.)

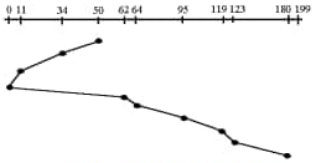
- **Disk-arm-scheduling** algorithms order pending accesses to tracks so that disk arm movement is minimized: Example: *Queue 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199*



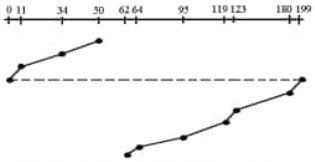
First Come -First Serve (FCFS)
640 Tracks, Oscillations



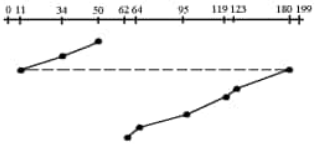
Shortest Seek Time First (SSTF)
236 Tracks, Starvation



Elevator (SCAN)
230 Tracks



Circular Scan (C-SCAN)
187 Tracks



C-LOOK
157 Tracks, Optimal

RAID

- **RAID: Redundant Arrays of Independent Disks**
  - disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
    - high capacity and high speed by using multiple disks in parallel,
    - high reliability by storing data redundantly, so that data can be recovered even if a disk fails
- The chance that some disk out of a set of *N* disks will fail is much higher than the chance that a specific single disk will fail
  - E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
  - Techniques for using redundancy to avoid data loss are critical with large numbers of disks
- Originally a cost-effective alternative to large, expensive disks
  - I in RAID originally stood for "inexpensive"
  - Today RAIDs are used for their higher reliability and bandwidth
    - The "I" is interpreted as independent

Improvement of Reliability via Redundancy

- **Bit-level striping** – split the bits of each byte across multiple disks
  - In an array of eight disks, write bit *i* of each byte to disk *i*
  - Each access can read data at eight times the rate of a single disk
  - But seek/access time worse than for a single disk
    - Bit level striping is not used much any more
- **Block-level striping** – with *n* disks, block *i* of a file goes to disk (*i* mod *n*) + 1
  - Requests for different blocks can run in parallel if the blocks reside on different disks
  - A request for a long sequence of blocks can utilize all disks in parallel

Improvement of Reliability via Redundancy

- **Bit-Interleaved Parity** – a single parity bit is enough for error correction, not just detection, since we know which disk has failed
  - When writing data, corresponding parity bits must also be computed and written to a parity bit disk
  - To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)
- **Block-Interleaved Parity**: Uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from *N* other disks
  - When writing data block, corresponding block of parity bits must also be computed and written to parity disk
  - To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.

Choice of RAID Level

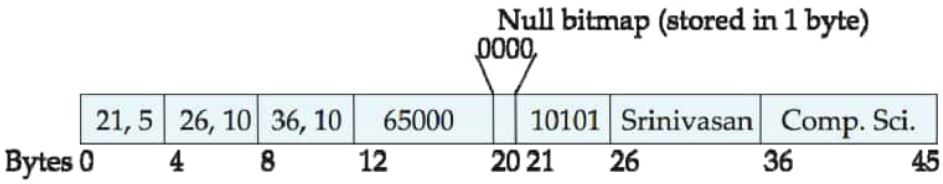
- Factors in choosing RAID level
  - Monetary cost
  - Performance: Number of I/O operations per second, and bandwidth during normal operation
  - Performance during failure
  - Performance during rebuild of failed disk
    - Including time taken to rebuild failed disk
- RAID 0 is used only when data safety is not important
  - E.g. data can be recovered quickly from other sources
- Level 2 and 4 never used since they are subsumed by 3 and 5
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (level 5) avoids
- Level 6 is rarely used since levels 1 and 5 offer adequate safety for most applications

Choice of RAID Level (Cont.)

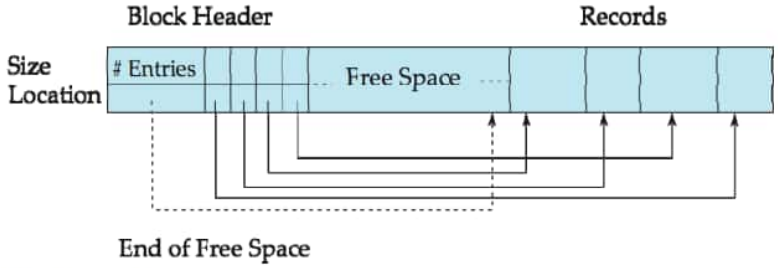
- Level 1 provides much better write performance than level 5
  - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
  - Level 1 preferred for high update environments such as log disks
- Level 1 had higher storage cost than level 5
  - disk drive capacities increasing rapidly (50%/year) whereas disk access times have decreased much less (x 3 in 10 years)
  - I/O requirements have increased greatly, e.g. for Web servers
  - When enough disks have been bought to satisfy required rate of I/O, they often have spare storage capacity
    - so there is often no extra monetary cost for Level 1!
- Level 5 is preferred for applications with low update rate, and large amounts of data
- Level 1 is preferred for all other applications

Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields (used in some older data models)
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap



Variable-Length Records: Slotted Page Structure



- **Slotted page** header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated
- Pointers should not point directly to record — instead they should point to the entry for the record in header

Storage Access

- A database file is partitioned into fixed-length storage units called **blocks**
  - Blocks are units of both storage allocation and data transfer
- Database system seeks to minimize the number of block transfers between the disk and memory
  - We can reduce the number of disk accesses by keeping as many blocks as possible in main memory
- **Buffer** – portion of main memory available to store copies of disk blocks
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory

Buffer Manager

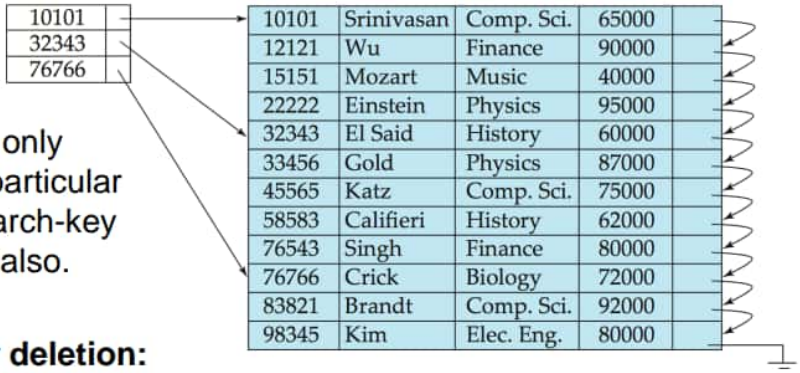
- Programs call on the buffer manager when they need a block from disk.
  1. If the block is already in the buffer, buffer manager returns the address of the block in main memory
  2. If the block is not in the buffer, the buffer manager
     1. Allocates space in the buffer for the block
        1. Replacing (throwing out) some other block, if required, to make space for the new block
        2. Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk
     2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester

Buffer-Replacement Policies (Cont.)

- **Pinned block** – memory block that is not allowed to be written back to disk
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Buffer managers also support **forced output** of blocks for the purpose of recovery

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value. For example, author catalog in library
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file
  - Also called non-clustering index
- Index-sequential file: ordered sequential file with a primary index



- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- **Single-level index entry deletion:**
  - **Dense indices** – deletion of search-key is similar to file record deletion
  - **Sparse indices** –
    - If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order)
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced

Index Update: Insertion

Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value appearing in the record to be inserted
  - **Dense indices** – if the search-key value does not appear in the index, insert it
  - **Sparse indices –** if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

Search Data Structures

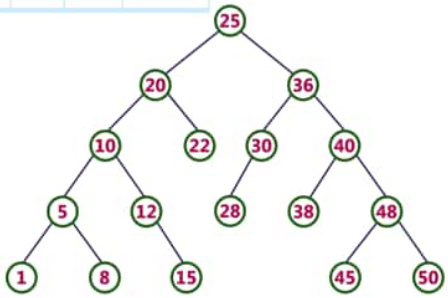
- How to search a key in a list of n data items?
  - Linear Search: O(n): Find 28 ➔ 16 comparisons
    - Unordered items in an array – search sequentially
    - Unordered / Ordered items in a list – search sequentially

22	50	20	36	40	15	08	01	45	48	30	10	38	12	25	28	05	END

  - Binary Search: O($\log_2$ n): Find 28 ➔ 4 comparisons – 25, 36, 30, 28
    - Ordered items in an array – search by divide-and-conquer

01	05	08	10	12	15	20	22	25	28	30	36	38	40	45	48	50	END

    - Binary Search Tree – recursively on left / right



PPD

Search Data Structures

- Worst case time (n data items in the data structure):

Data Structure	Search	Insert	Delete	Remarks
Unordered Array	O(n)	O(1)	O(1)	The time to Insert / Delete an item is the time after the location of the item has been ascertained by Search.
Ordered Array	O(log n)	O(n)	O(n)	
Unordered List	O(n)	O(1)	O(1)	
Ordered List	O(n)	O(1)	O(1)	
Binary Search Tree	O(h)	O(1)	O(1)	

- Between an array and a list, there is a trade-off between search and insert/delete complexity
- For a BST of n nodes, log n <= h <= n, where h is the height of the tree
- A BST is balanced if h ~ O(log n) – this what we desire

Balanced Binary Search Trees

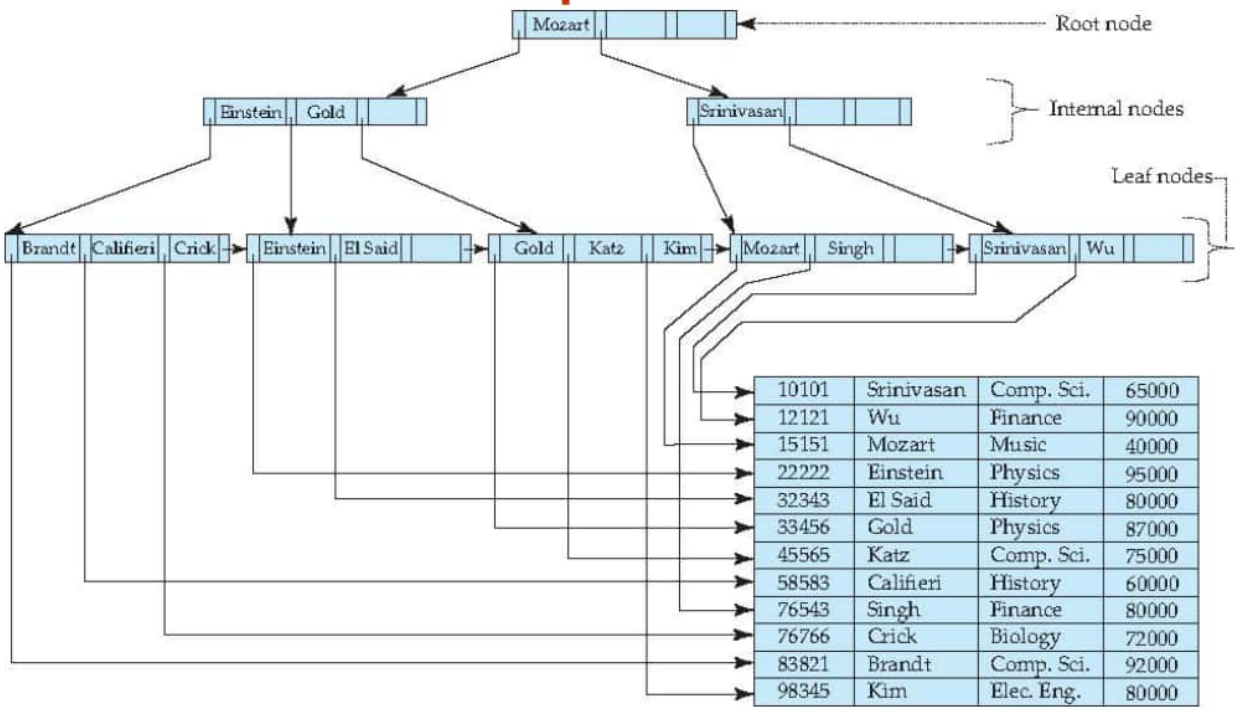
- A BST is balanced if h ~ O(log n)
- Balancing guarantees may be of various types:
  - Worst-case
    - AVL Tree
  - Randomized
    - Randomized BST, Skip List
  - Amortized
    - Splay
- These data structures have optimal complexity for all of search, insert and delete – O(log n). However:
  - Good for in memory operations
  - Work well for small volume of data
  - Has complex rotation and / or similar operations
  - Do not scale for external data structures

PPD

2-3-4 Trees

- Consider only one node type with space for 3 items and 4 links
  - Internal node (non-root) has 2 to 4 children (links)
  - Leaf node has 1 to 3 items
  - Wastes some space, but has several advantages for external data structure
- Generalizes easily to larger nodes
  - All paths from root to leaf are of the same length
  - Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
  - A leaf node has between $\lceil (n-1)/2 \rceil$ and n–1 values
  - Special cases:
    - If the root is not a leaf, it has at least 2 children.
    - If the root is a leaf, it can have between 0 and (n–1) values.
- Extends to external data structures
  - B-Tree
  - 2-3-4 Tree is a B-Tree where n = 4

Example of B+-Tree



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B+-Tree Index Files (Cont.)

A B+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.

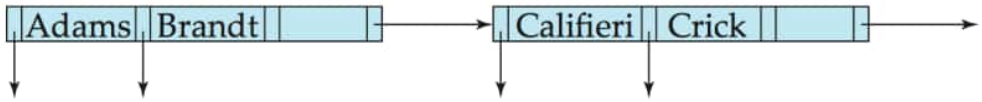
** Errata note: modified find procedure missing in first printing of 6th edition

Queries on B+-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry)
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1,000,000) = 4$ nodes are accessed in a lookup
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Updates on B+-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node
  - let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split
  - If the parent is full, split it and **propagate** the split further up
- Splitting of nodes proceeds upwards till a node that is not full is found
  - In the worst case the root node may be split increasing the height of the tree by 1



Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range — typically b-bit integers, with $b = 32$
  - At any time use only a prefix of the hash function to index into a table of bucket addresses
  - Let the length of the prefix be i bits, $0 \le i \le 32$
    - Bucket address table size = 2^i. Initially $i = 0$
    - Value of i grows and shrinks as the size of the database grows and shrinks
  - Multiple entries in the bucket address table may point to a bucket (why?)
  - Thus, actual number of buckets is $< 2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key
  - If range queries are common, ordered indices are to be preferred
- **In practice**:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

- Counting number of matching tuples is even faster

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - **Existence bitmap** to note if there is a valid record at a record location
  - Needed for complementation
    - not(*A*=*v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(*A*=*v*):
    - intersect above result with (NOT *bitmap-A-Null*)

PPD

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name> (<attribute-list>)

  E.g.: **create index** *b-index* **on** *branch(branch_name)*
- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key
  - Not really required if SQL **unique** integrity constraint is supported – it is preferred
- To drop an index

  drop index <index-name>
- Most database systems allow specification of type of index, and clustering
  - You can also create an index for a cluster
  - You can create a composite index on multiple columns up to a maximum of 32 columns
    - A composite index key cannot exceed roughly one-half (minus some overhead) of the available space in the data block

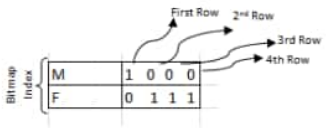
Source: https://docs.oracle.com/cd/B10500_01/appdev.920/a96590/adg0Gidx.htm

PPD

Bitmap Index in SQL

- **create bitmap index** <index-name> **on** <relation-name>(<attribute-list>)
- **Example:**
  - Student (Student_ID, Name, Address, Age, Gender, Semester)
  - CREATE BITMAP INDEX Idx_Gender ON Student (Gender);
  - CREATE BITMAP INDEX Idx_ Semester ON Student (Semester);

STUDENT					
STUDENT_ID	STUDENT_NAME	ADDRESS	AGE	GENDER	SEMESTER
100	Joseph	Alaiedon Township	20	M	1
101	Allen	Fraser Township	21	F	1
102	Chris	Clinton Township	20	F	2
103	Patty	Troy	22	F	4



SEMESTER	
1	1 1 0 0
2	0 0 1 0
3	0 0 0 0
4	0 0 0 1

- SELECT * FROM Student WHERE Gender = 'F' AND Semester =4;
  - AND 0 1 1 1 with 0 0 0 1 to get the result

- Adjust the indexes on the tables to optimize performance

- There is no sound theory that determines optimal performance. Rather, we take a quick look into a few common guidelines that can help you keep your database agile in its behavior

Guidelines for Indexing

- **Rule 0: Indexes lead to Access – Update Tradeoff**
  - Every query (access) results in a 'search' on the underlying physical data structures
    - Having specific index on search field can significantly improve performance
  - Every update (insert / delete / values update) results in update of the index files – an overhead or penalty for quicker access
    - Having unnecessary indexes can cause significant degradation of performance of various operations
    - Index files may also occupy significant space on your disk and / or
    - Cause slow behavior due to memory limitations during index computations
  - Use informed judgment to index!

PPD

Guidelines for Indexing

- **Rule 1: Index the Correct Tables**
  - Create an index if you frequently want to **retrieve less than 15%** of the rows in a large table
    - The percentage varies greatly according to the relative speed of a table scan and how clustered the row data is about the index key
      - The faster the table scan, the lower the percentage
      - More clustered the row data, the higher the percentage
  - Index columns used for joins to improve performance on **joins of multiple tables**
  - Primary and unique keys automatically have indexes, but you might want to create an **index on a foreign key**
  - **Small tables** do not require indexes
    - If a query is taking too long, then the table might have grown from small to large

- **Rule 2: Index the Correct Columns**
  - Columns with one or more of the following characteristics are candidates for indexing:
    - Values are relatively unique in the column
    - There is a wide range of values (good for regular indexes)
    - There is a small range of values (good for bitmap indexes)
    - The column contains many nulls, but queries often select all rows having a value. In this case, a comparison that matches all the non-null values, such as:
      - WHERE COL_X > -9.99 *power(10,125) is preferable to WHERE COL_X IS NOT NULL
      - This is because the first uses an index on COL_X (if COL_X is a numeric column)
  - Columns with the following characteristics are less suitable for indexing:
    - There are many nulls in the column and you do not search on the non-null values
    - LONG and LONG RAW columns cannot be indexed
  - The size of a single index entry cannot exceed roughly one-half (minus some overhead) of the available spa

PPD

Guidelines for Indexing

- **Rule 3: Limit the Number of Indexes for Each Table**
  - The more indexes, the more overhead is incurred as the table is altered
    - When rows are inserted or deleted, all indexes on the table must be updated
    - When a column is updated, all indexes on the column must be updated
  - You must weigh the performance benefit of indexes for queries against the performance overhead of updates
    - If a table is primarily read-only, you might use more indexes; but, if a table is heavily updated, you might use fewer indexes

PPD

Guidelines for Indexing

Table VENDOR_PARTS

VEND ID	PART NO	UNIT COST
1012	10–440	.25
1012	10–441	.39
1012	457	4.95
1010	10–440	.27
1010	457	5.10
1220	08–300	1.33
1012	08–300	1.19
1292	457	5.28

- **Rule 4: Choose the Order of Columns in Composite Indexes**
  - The order of columns in the CREATE INDEX statement can affect performance
    - Put the column expected to be used most often first in the index
    - You can create a composite index (using several columns), and the same index can be used for queries that reference all of these columns, or just some of them
  - For the VENDOR_PARTS table, assume that there are 5 vendors, and each vendor has about 1000 parts. Suppose VENDOR_PARTS is commonly queried as:
    - SELECT * FROM vendor_parts WHERE part_no = 457 AND vendor_id = 1012;
    - Create a composite index with the most selective (with most values) column first
      - CREATE INDEX ind_vendor_id ON vendor_parts (part_no, vendor_id);
  - Composite indexes speed up queries that use the leading portion of the index:
    - So queries with WHERE clauses using only PART_NO column also runs faster
    - With only 5 distinct values, a separate index on VENDOR_ID does not help

Source: https://docs.oracle.com/cd/B10500_01/appdev.920/a96590/adg06idx.htm

PPD

Guidelines for Indexing

- **Rule 5: Gather Statistics to Make Index Usage More Accurate**
  - The database can use indexes more effectively when it has statistical information about the tables involved in the queries
    - Gather statistics when the indexes are created by including the keywords COMPUTE STATISTICS in the CREATE INDEX statement
    - As data is updated and the distribution of values changes, periodically refresh the statistics by calling procedures like (in Oracle):
      - DBMS_STATS.GATHER_TABLE_STATISTICS and
      - DBMS_STATS.GATHER_SCHEMA_STATISTICS

Source: https://docs.oracle.com/cd/B10500_01/appdev.920/a96590/adg06idx.htm

PPD

PPD

Guidelines for Indexing

- **Rule 6: Drop Indexes That Are No Longer Required**
  - You might drop an index if:
    - It does not speed up queries. The table might be very small, or there might be many rows in the table but very few index entries
    - The queries in your applications do not use the index
    - The index must be dropped before being rebuilt
  - When you drop an index, all extents of the index's segment are returned to the containing tablespace and become available for other objects in the tablespace
  - Use the SQL command DROP INDEX to drop an index. For example, the following statement drops a specific named index:
    - DROP INDEX Emp_ename;
  - If you drop a table, then all associated indexes are dropped
  - To drop an index, the index must be contained in your schema or you must have the DROP ANY INDEX system privilege

Source: https://docs.oracle.com/cd/B10500_01/appdev.920/a96590/adg06idx.htm

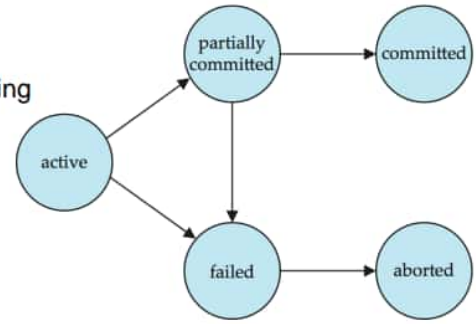
ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity:**
  - Either all operations of the transaction are properly reflected in the database or none are
- **Consistency:**
  - Execution of a transaction in isolation preserves the consistency of the database
- **Isolation:**
  - Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished
- **Durability:**
  - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

Transaction State

- **Active**
  - The initial state; the transaction stays in this state while it is executing
- **Partially committed**
  - After the final statement has been executed
- **Failed**
  - After the discovery that normal execution can no longer proceed
- **Aborted**
  - After the transaction has been rolled back and the database 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



Transaction to transfer \$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)

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction
- In SQL, a transaction begins implicitly
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one
  - **Rollback work** causes current transaction to abort
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - For example in JDBC, connection.setAutoCommit(false);

PPD

Transaction Control Language (TCL)

- The following commands are used to control transactions.
  - **COMMIT** − to save the changes
  - **ROLLBACK** − to roll back the changes
  - **SAVEPOINT** − creates points within the groups of transactions in which to ROLLBACK
  - **SET TRANSACTION** − Places a name on a transaction
- Transactional control commands are only used with the **DML Commands** such as
  - INSERT, UPDATE and DELETE only
  - They cannot be used while creating tables or dropping them because these operations are automatically committed in the database

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.
  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 .
  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'
- 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
- Below is a schedule which is view-serializable but *not* conflict serializable

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
  - T_{27}–T_{28}–T_{29}
  - The one read(Q) instruction reads the initial value of Q in both schedules and
  - T_{29} performs the final write of Q in both schedules
- T_{28} and T_{29} perform write(Q) operations called **blind writes**, without having performed a read(Q) operation
- Every view serializable schedule that is not conflict serializable has **blind writes**

View Serializability: Example 1

- Check whether the schedule is view serializable or not?
  - S : R2(B); R2(A); R1(A); R3(A); W1(B); W2(B); W3(B);
- Solution:
  - With 3 transactions, total number of schedules possible = 3! = 6
    - <T1 T2 T3>
    - <T1 T3 T2>
    - <T2 T3 T1>
    - <T2 T1 T3>
    - <T3 T1 T2>
    - <T3 T2 T1>
  - Final update on data items :
    - A : -
    - B : T1 T2 T3
    - Since the final update on B is made by T3, so the transaction T3 must execute after transactions T1 and T2.
    - Therefore, (T1,T2) → T3. Now, Removing those schedules in which T3 is not executing at last:
      - <T1 T2 T3>
      - <T2 T1 T3>

Source: http://www.edugrabs.com/how-to-check-for-view-serializable-schedule/

Source: http://www.edugrabs.com/how-to-check-for-view-serializable-schedule/

View Serializability: Example 1

- Check whether the schedule is view serializable or not?
  - S : R2(B); R2(A); R1(A); R3(A); W1(B); W2(B); W3(B);
- Solution:
  - Initial Read + Which transaction updates after read?
    - A : T2 T1 T3 (initial read)
    - B : T2 (initial read); T1 (update after read)
    - The transaction T2 reads B initially which is updated by T1. So T2 must execute before T1.
    - Hence, T2 → T1. Removing those schedules in which T2 is executing before T1:
    - <T2 T1 T3>
  - Write Read Sequence (WR)
    - No need to check here
  - Hence, view equivalent serial schedule is:
    - **T2 → T1 → T3**

More Complex Notions of Serializability

- The schedule below produces the same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it

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

- If we start with A = 1000 and B = 2000, the final result is 960 and 2040
- Determining such equivalence requires analysis of operations other than read and write

manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item

- Should a transaction hold a lock on the whole database
  - Would lead to strictly serial schedules – very poor performance

- The most common method used to implement locking requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item

- Concurrency Control
- **Lock-Based Protocols**
- Implementing Locking

Lock-Based Protocols

- 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
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction
- A transaction can unlock a data item *Q* by the **unlock**(*Q*) Instruction
- Lock requests are made to the concurrency-control manager by the programmer
- Transaction can proceed only after request is granted

Lock-Based Protocols: Example

- Given, T3 and T4, consider Schedule 2 (partial)
- Since *T*3 is holding an exclusive mode lock on *B* and *T*4 is requesting a shared-mode lock on *B*, *T*4 is waiting for *T*3 to unlock *B*
- Similarly, since *T*4 is holding a shared-mode lock on *A* and *T*3 is requesting an exclusive-mode lock on *A*, *T*3 is waiting for *T*4 to unlock *A*
- Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution
- This situation is called **deadlock**
- When deadlock occurs, the system must roll back one of the two transactions.
- Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked
- These data items are then available to the other transaction, which can continue with its execution

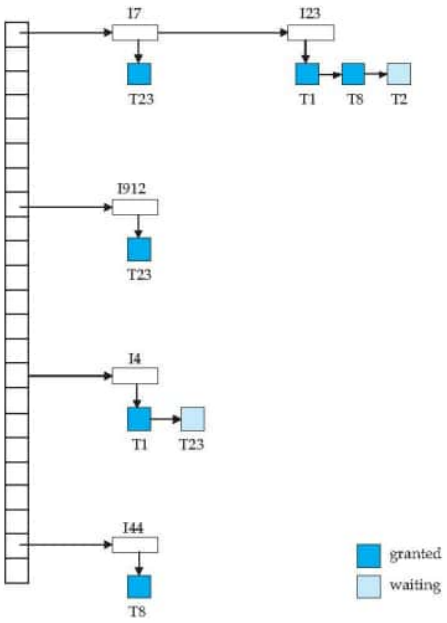
*T*3:	*T*4:
lock-X(*B*);	lock-S(*A*);
read(*B*);	read(*A*);
B := *B* − 50;	lock-S(*B*);
write(*B*);	read(*B*);
lock-X(*A*);	display(*A* + *B*);
read(*A*);	unlock(*A*);
A := *A* + 50;	unlock(*B*)
write(*A*);	
unlock(*B*);	
unlock(*A*)	

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

More Two Phase Locking Protocols

- To avoid Cascading roll-back, follow a modified protocol called **strict two-phase locking**
  - a transaction must hold all its exclusive locks till it commits/aborts
- **Rigorous two-phase locking** is even stricter.
  - *All* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit

Lock Table



- Dark blue rectangles indicate granted locks; light blue indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

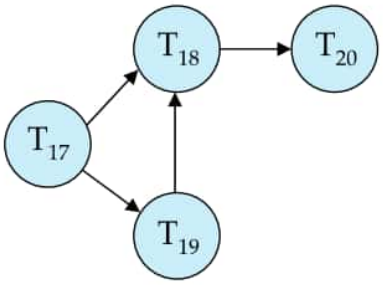
Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

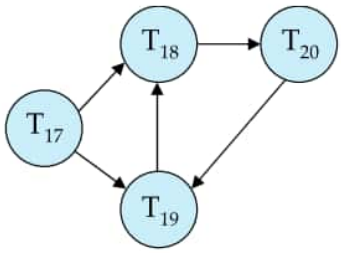
Deadlock Prevention

- Following schemes use transaction timestamps for the sake of deadlock prevention alone
- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item. (older means smaller timestamp)
    - Younger transactions never wait for older ones; they are rolled back instead
  - A transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it
    - Younger transactions may wait for older ones
  - May be fewer rollbacks than *wait-die* scheme

Deadlock Detection: Example



Wait-for graph without a cycle



Wait-for graph with a cycle

Timestamp-Based Protocols

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order
- 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)).

Timestamp-Based Protocols (Cont.)

- 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)

Undo and Redo Operations

- **Undo** of a log record $<T_i, X, V_1, V_2>$ writes the **old** value V_1 to X
- **Redo** of a log record $<T_i, X, V_1, V_2>$ writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - Each time a data item X is restored to its old value V a special log record (called **redo-only**) $<T_i, X, V>$ is written out
    - When undo of a transaction is complete, a log record $<T_i$ **abort**> is written out (to indicate that the undo was completed)
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - No logging is done in this case

Undo and Redo on Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record <T_i **start**>,
    - but does not contain either the record <T_i **commit**> *or* <T_i **abort**>
  - Transaction T_i needs to be redone if the log
    - contains the records <T_i **start**>
    - and contains the record <T_i **commit**> *or* <T_i **abort**>
  - It may seem strange to redo transaction T_i if the record <T_i **abort**> record is in the log
    - To see why this works, note that if <T_i **abort**> is in the log, so are the redo-only records written by the undo operation. Thus, the end result will be to undo T_i 's modifications in this case. This slight redundancy simplifies the recovery algorithm and enables faster overall recovery time
    - such a redo redoes all the original actions including the steps that restored old value – Known as **repeating history**

Immediate Modification Recovery Example

Immediate Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit>

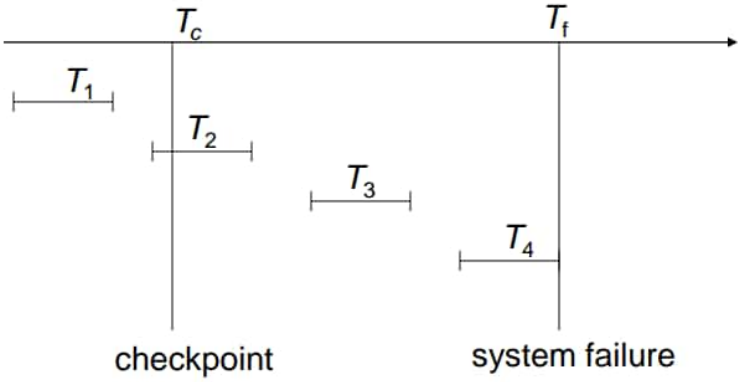
Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records $<T_0$, B, 2000>, $<T_0$, A, 1000>, $<T_0$, **abort**> are written out

(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records $<T_1$, C, 700>, $<T_1$, **abort**> are written out

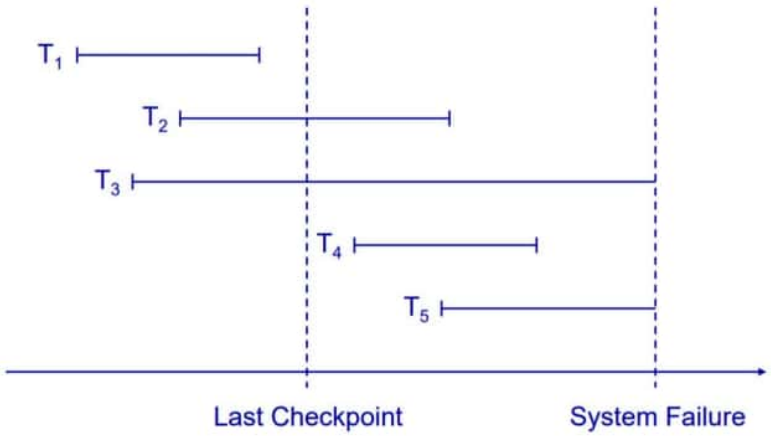
(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Example of Checkpoints



- Any transactions that committed before the last checkpoint should be ignored
  - T_1 can be ignored (updates already output to disk due to checkpoint)
- Any transactions that committed since the last checkpoint need to be redone
  - T_2 and T_3 redone
- Any transaction that was running at the time of failure needs to be undone and restarted
  - T_4 undone

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions



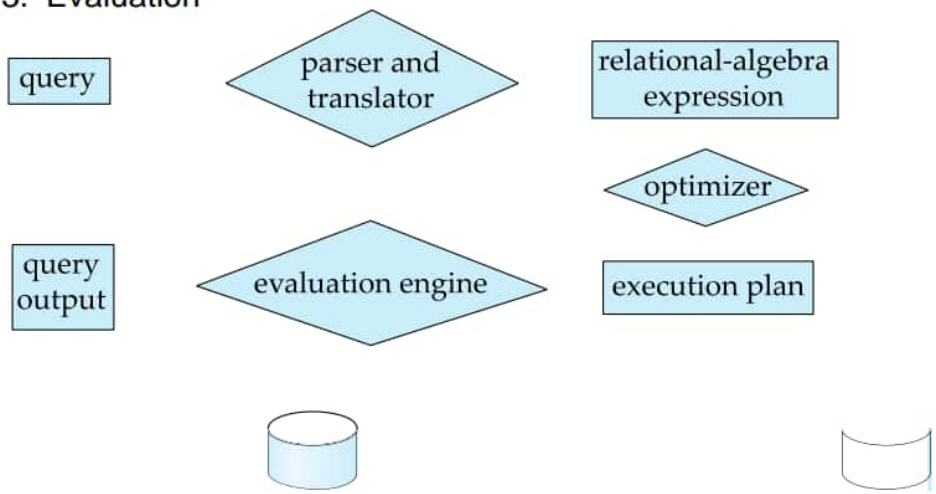
Requirement:
- Transactions of type T1 need no recovery
- Transactions of type T2 or T4 need to be redone
- Transactions of type T3 or T5 need to be undone and restarted

Strategy:
- Ignore T1
- Redo T2, T3, T4 and T5
- Undo T3 and T5

Recovery Algorithm (Cont.)

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *salary* to find instructors with salary < 75000,
  - or can perform complete relation scan and discard instructors with salary ≥ 75000

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate
- Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful

Measures of Query Cost (Cont.)

- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks
    $b * t_T + S * t_S$
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae

Selection Operation

- **File scan**
- Algorithm **A1** (**linear search**). Scan each file block and test all records to see whether they satisfy the selection condition
  - Cost estimate = b_r block transfers + 1 seek
    - b_r denotes number of blocks containing records from relation r
  - If selection is on a key attribute, can stop on finding record
    - cost = (b_r/2) block transfers + 1 seek
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices
- Note: binary search generally does not make sense since data is not stored consecutively
  - except when there is an index available,
  - and binary search requires more seeks than index search

- except when there is an index available,
- and binary search requires more seeks than index search

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index
  - h_i = height of B+ Tree
- **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
- **A3** (**primary index, equality on nonkey**) Retrieve multiple records
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

- To compute the theta join $r \bowtie_\theta s$

```
for each tuple t_r in r do begin
  for each tuple t_s in s do begin
      test pair (t_r, t_s) to see if they satisfy the join condition θ
      if they do, add t_r • t_s to the result.
  end
end
```

- r is called the **outer relation** and s the **inner relation** of the join
- Requires no indices and can be used with any kind of join condition
- Expensive since it examines every pair of tuples in the two relations

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  $n_r * b_s + b_r$ block transfers, plus
  $n_r + b_r$ seeks
- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- Example of join of *students* and *takes*:
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400
- Assuming worst case memory availability cost estimate is
  - with *student* as outer relation:
    - 5000 * 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 * 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers
- Block nested-loops algorithm (next slide) is preferable

Block Nested-Loop Join

Block Nested-Loop Join (Cont.)

- Worst case estimate: $b_r * b_s + b_r$ block transfers + 2 * b_r seeks
  - Each block in the inner relation *s* is read once for each *block* in the outer relation
- Best case: $b_r + b_s$ block transfers + 2 seeks.
- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use M — 2 disk blocks as blocking unit for outer relations, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M-2) \rceil$ seeks
  - If equi-join attribute forms a key or inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available (next slide)

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins
   a. $\sigma_{\theta}(E_1 \text{ X } E_2) = E_1 \bowtie_{\theta} E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. (a) Natural join operations are associative:

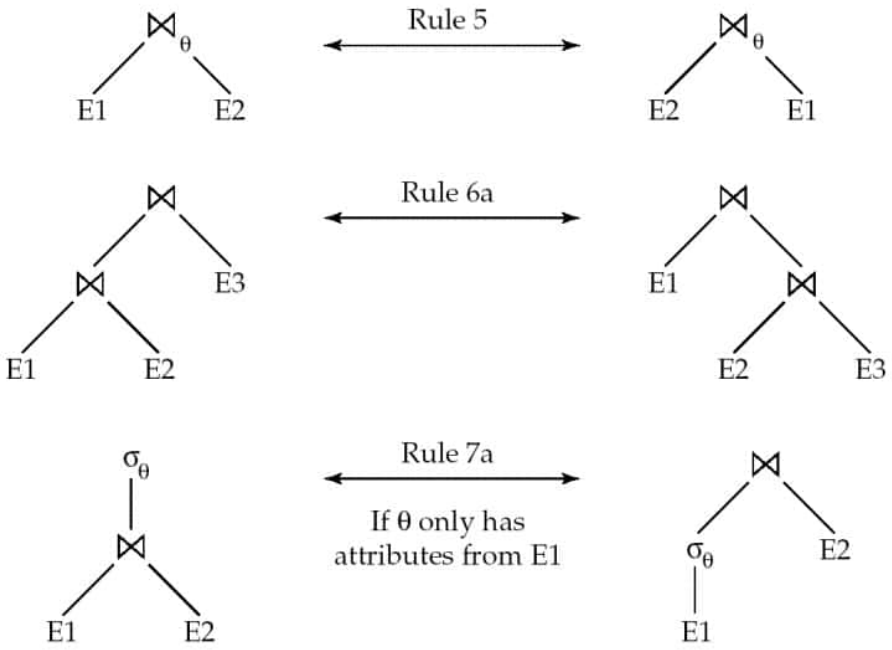
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

   - (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of $-$

Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$

and similarly for $\cap$ in place of $-$, but not for $\cup$

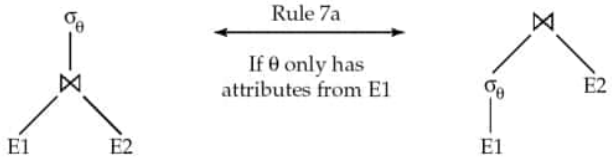
12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Transformation Example: Pushing Selections

- Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"}}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$
- Transformation using rule 7a
  - $\Pi_{name,\ title}((\sigma_{dept\_name=\text{"Music"}}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course)))$
- Performing the selection as early as possible reduces the size of the relation to be joined

course(*course id*, *title*, *dept name*, *credits*)
instructor(*ID*, *name*, *dept name*, *salary*)
teaches(*ID*, *course id*, *sec id*, *semester*, *year*)



What About Smaller Schemas?

- Suppose we had started with *inst_dept*. How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule "if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key"
- Denote as a **functional dependency**:

  $$dept\_name \rightarrow building, budget$$

- In *inst_dept*, because *dept_name* is not a candidate key, the building and budget of a department may have to be repeated.
  - This indicates the need to decompose *inst_dept*
- Not all decompositions are good. Suppose we decompose *employee(ID, name, street, city, salary)* into

  employee1 (*ID*, *name*)

  employee2 (*name*, *street, city, salary*)
- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

A Combined Schema Without Repetition

- Consider combining relations
  - *sec_class(sec_id, building, room_number)* and
  - *section(course_id, sec_id, semester, year)*

  into one relation
  - *section(course_id, sec_id, semester, year, building, room_number)*
- No repetition in this case

Example of Lossless-Join Decomposition

- **Lossless join decomposition**
- Decomposition of $R = (A, B, C)$
  $R_1 = (A, B)$ $R_2 = (B, C)$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

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

B	C
1	A
2	B

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

$\Pi_A (r) \bowtie \Pi_B (r)$

A	B	C
α	1	A
β	2	B

First Normal Form (1NF)

PPD

- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in **first normal form** if
  - the domains of all attributes of R are atomic
  - the value of each attribute contains only a single value from that domain
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form

Functional Dependencies (Cont.)

- Let R be a relation schema

  $$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- 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,

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

- Example: Consider r(A,B) with the following instance of r.

1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F
  - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F
- We denote the *closure* of F by **F^+**
- F^+ is a superset of F
  - $F = \{A \rightarrow B, B \rightarrow C\}$
  - $F^+ = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$

Boyce-Codd Normal Form

A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for R

Example schema *not* in BCNF:

instr_dept (*ID, name, salary, dept_name, building, budget*)

because *dept_name* → *building, budget* holds on *instr_dept*, but *dept_name* is not a superkey

BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving*.
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form*.

Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

  $$\alpha \rightarrow \beta \text{ in } F^+$$

  at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for R
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R

    (**NOTE**: each attribute may be in a different candidate key)
- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold)
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later)

Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F
  - For e.g.: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F
- We denote the *closure* of F by F^+

Closure of a Set of Functional Dependencies

- We can find F^+, the closure of F, by repeatedly applying **Armstrong's Axioms:**
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
  - if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ **(augmentation)**
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
  - **sound** (generate only functional dependencies that actually hold), and
  - **complete** (generate all functional dependencies that hold)

Example

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- Some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

Closure of Functional Dependencies (Cont.)

- Additional rules:
  - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds **(union)**
  - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)**
  - If $\alpha \rightarrow \beta$ holds and $\gamma\ \beta \rightarrow \delta$ holds, then $\alpha\ \gamma \rightarrow \delta$ holds **(pseudotransitivity)**

  The above rules can be inferred from Armstrong's axioms

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$
- Is AG a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C, A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g.: on RHS: $\{\boldsymbol{A \rightarrow B}, \quad \boldsymbol{B \rightarrow C}, \quad \boldsymbol{A \rightarrow CD}\}$ can be simplified to $\{\boldsymbol{A \rightarrow B}, \quad \boldsymbol{B \rightarrow C}, \quad \boldsymbol{A \rightarrow D}\}$
      - *In the forward: (1)* $A \rightarrow CD \Rightarrow A \rightarrow C$ *and* $A \rightarrow D$ *(2)* $A \rightarrow B, \quad B \rightarrow C \Rightarrow A \rightarrow C$
      - *In the reverse: (1)* $A \rightarrow B, \quad B \rightarrow C \Rightarrow A \rightarrow C$ *(2)* $A \rightarrow C, A \rightarrow D \Rightarrow A \rightarrow CD$
    - E.g.: on LHS: $\{\mathbf{A \rightarrow B}, \quad \boldsymbol{B \rightarrow C}, \quad \boldsymbol{AC \rightarrow D}\}$ can be simplified to $\{\mathbf{A \rightarrow B}, \quad \boldsymbol{B \rightarrow C}, \quad \boldsymbol{A \rightarrow D}\}$
      - *In the forward: (1)* $A \rightarrow B, \quad B \rightarrow C \Rightarrow A \rightarrow C \Rightarrow A \rightarrow AC$ *(2)* $A \rightarrow AC$ *,* $AC \rightarrow D \Rightarrow A \rightarrow D$
      - *In the reverse:* $A \rightarrow D \Rightarrow AC \rightarrow D$
- Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

Extraneous Attributes

- Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.
  - Attribute A is **extraneous** in α if $A \in \alpha$
    and *F* logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - Attribute *A* is **extraneous** in β if $A \in \beta$
    and the set of functional dependencies
    $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies *F*.
- *Note:* Implication in the opposite direction is trivial in each of the cases above, since a "stronger" functional dependency always implies a weaker one
- Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
  - *B* is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (I.e. the result of dropping *B* from $AB \rightarrow C$).
  - A+ = AC in $\{A \rightarrow C, AB \rightarrow C\}$
- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
  - *C* is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting *C*
  - *AB+ = ABCD in* $\{A \rightarrow C, AB \rightarrow D\}$

Computing a Canonical Cover

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
    - Yes: in fact, $B \rightarrow C$ is already present!
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
    - Yes: using transitivity on $A \rightarrow B$ *and* $B \rightarrow C$.
      - Can use attribute closure of A in more complex cases
- The canonical cover is: $A \rightarrow B$
  $B \rightarrow C$

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \prod_{R1}(r) \bowtie \prod_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

To Identify whether a decomposition is lossy or lossless, it must satisfy the following conditions :

- $R_1 \cup R_2 = R$
- $R_1 \cap R_2 \neq \Phi$ *and*
- $R_1 \cap R_2 \rightarrow R_1$ *or* $R_1 \cap R_2 \rightarrow R_2$

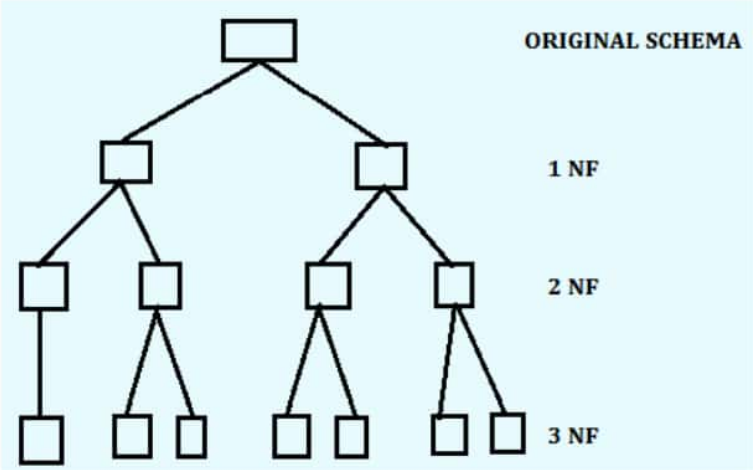
Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
  - Can be decomposed in two different ways
- $R_1 = (A, B)$, $R_2 = (B, C)$
  - Lossless-join decomposition:

    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
  - Dependency preserving
- $R_1 = (A, B)$, $R_2 = (A, C)$
  - Lossless-join decomposition:

    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

- Lossless Join Decomposition Property
  - It should be possible to reconstruct the original table
- Dependency Preserving Property
  - No functional dependency (or other constraints should get violated)



Third Normal Form: Motivation

- There are some situations where
  - BCNF is not dependency preserving, and
  - Efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form (3NF)
  - Allows some redundancy (with resultant problems; as seen above)
  - But functional dependencies can be checked on individual relations without computing a join
  - **There is always a lossless-join, dependency-preserving decomposition into 3NF**

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - the decomposition is lossless
  - it may not be possible to preserve dependencies.

S#	3NF	BCNF
1.	It concentrates on Primary Key	It concentrates on Candidate Key.
2.	Redundancy is high as compared to BCNF	0% redundancy
3.	It may preserve all the dependencies	It may not preserve the dependencies.
4.	A dependency X → Y is allowed in 3NF if X is a super key or Y is a part of some key.	A dependency X → Y is allowed if X is a super key

Fourth Normal Form

- A relation schema R is in **4NF** with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
  - α is a superkey for schema R
- If a relation is in 4NF it is in BCNF