CSE132A Final Review

This review doc includes almost everything from the lectures. It also covers some exemplary questions in the practice homework and sample final found online. We are also making a collaborative **cheatsheet**.
Created by M. Free feel to collaborate. Try using Chrome's "Clock Block" add-on to format codes.

Logistics

- Topics: recursive queries, relational algebra, query processing (includes join minimization), schema design and concurrency control (up tof and excluding shared locks)
- Format: There will be some T/F questions, problems on join minimization with FDs and schema design similar to the homework, and questions on recursive queries, relational algebra, and concurrency control.
- Date: **March 21, 7pm, York 2622**
- Cheatsheet (A4, one sided) is allowed.

LEC 10 - Recursion

- Expressivity of SQL
  - Full programming languages (Java, C, etc.) can express all computable functions
  - *Theory of Computation*: SQL can only use polynomial space which is bounded by the original database. Low complexity relative to space → Limitation

- **Transitive Closure of a Graph**

  - E.g. Is there a way to get from city1 to city 2 directly/with at most one stopover/with at most two stopover.

```
SELECT x.from, y.to
FROM flight x, flight y
WHERE x.from = "city1" AND x.to = y.from AND y.to = "city2"
```

    - SQL can't express arbitrary k times of stopover

  - Similar examples
    - Parts-components relation: find all subparts of some given part
    - Parent/child relation: find all of John's descendants (don't know how far down the tree)

  - **General: Transitive Closure of a Graph**
    - Find the pairs of nodes <x, y> that are connected by **some directed path**
    - **Distance(x, y):** length of shortest path
    - **Diameter(G):** length of the maximum distance

- E.g. Relation G contains pairs (A, B) as a path
  - Original graphs and adding “at distance k” until no new nodes are added. Distance cannot be larger than the total number of nodes in G.

- **Extension on SQL to include recursion**

  - Induction: denote by T_k the pairs of nodes at distance at most k.
    - Visual presentation
      A $\rightarrow^{X}_{G}$ Intermediate node $\rightarrow^{Y}_{Tk-1}$ B
    - T_1 = Find pairs of nodes <a, b> at distance 1

      ```
      SELECT * FROM G
      ```
    - T_k = Find the pairs of nodes <a, b> at distance at most k. If a can reach some intermediate nodes that is within T_{k-1} to b, then a can reach b at most distance k.

      ```
      SELECT * FROM Tk-1
      UNION
      SELECT x.A, y.B FROM G x, Tk-1 y
      WHERE x.B = y.A
      ```
    - **CAN’T** be done in the basic SQL

  - **Recursive View**
    - Semantics: define a view in terms of itself
      - Start with empty T
      - While T changes: evaluate view with current T union result with T
    - Code example: built in termination

      ```
      CREATE RECURSIVE VIEW T AS
      SELECT * FROM G
      UNION
      SELECT x.A, y.B
      FROM G x, T y
      WHERE x.B = y.A
      ```
    - Termination is *guaranteed* because there are finitely many tuples one can add to T (at most n^2 → Number of iterations: diameter of T).
    - *Two parts*: one depend on the T for recursive step. One doesn’t depend on T to initialize the T.

  - Alternative:
    - Computation is the same, but not kept as a view

      ```
      WITH RECURSIVE T AS
      ```

```
SELECT * FROM G
UNION
SELECT x.A, y.B
FROM G x, T y
WHERE x.B = y.A
```

- E.g. Friends: drinkers who frequent the same bar. Find transitive closure of friends. (typical social network queries)

Frequents	Drinkers	Bar
f1	**B**	**Space**
f2	C	**Space**

T	Drinker1	Drinker2
t1	A	**B**

```
CREATE RECURSIVE VIEW T as
SELECT f1.drinker as drinker1, f2.drinker as drinker2
FROM frequents f1, frequents f2
WHERE f1.bar = f2.bar
UNION
SELECT t1.drinker1, f2.drinker as drinker2
FROM T t1, frequents f1, frequents f2
WHERE t1.drinker2 = f1.drinker AND f1.bar = f2.bar
```

- Problematic Example
  - E.g. keep adding N + 1 from N. It never terminates.

```
CREATE RECURSIVE VIEW T as
SELECT a, 0 AS n FROM R
UNION
SELECT a, n + 1 as N FROM T
```

  - Most systems don't support numeric operations on the recursive query

- **Embedded SQL**

  - Client (Full programming language) + DB Server
    - Make queries from within the programming languages and receive answers and return.
    - Programming languages support loops, etc.

- Pseudo-code

```
T = G
Delta = G
while Delta != 0:
      Old_T = T
      T =
```

```
SELECT * FROM T
       UNION
SELECT x.A, y.B FROM G x, T y WHERE x.B = y.A
```

```
      Delta = T - Old_T
return T
```

- Runtime: converges in **diameter(G)** iterations (maximum distance between two nodes in G.

- **Optimization**

  - *Semi-naive:* instead of using T, we use Delta → use at least one new tuple every time

    ```
    SELECT * FROM T
           UNION
    SELECT x.A, y.B FROM G x, Delta y WHERE x.B = y.A
    ```

    - No guarantee that it eliminates all recomputation. But in practice is better.

  - *Double Recursion*: both new index comes from T.

    ```
    SELECT * FROM T
           UNION
    SELECT x.A, y.B FROM T x, T y WHERE x.B = y.A
    ```

    - Converges in **log(diameter(G))** iterations.
    - If use three T's, it change log base 2 to log base 3. Not a big gain.
    - It depends on the implementation whether to use the double recursion...

  - *Double Recursion with Semi-Naive*: use at least one new tuple every time.

```
SELECT * FROM T
       UNION
SELECT x.A, y.B FROM Delta x, T y WHERE x.B = y.A
       UNION
SELECT x.A, y.B FROM T x, Delta y WHERE x.B = y.A
```

LEC 11 - Query Processing Basics *Feb 12*

- Big Picture
  - SQL → Relational Algebra (procedure, provides some operations) → Query Rewriting (more efficient) → Query Execution Plan → Execution

- **Relational Algebra**
  As expressive as basic SQL (set semantics/no duplicates; no null)
  - Projection **π**: limit some columns of table; *Monotonic*
  - Selection **σ**: select some satisfying conditions; *Monotonic*
  - Join ⋈: natural join; *Monotonic*
  - Union **U** (*Monotonic)*, difference **-** (*Non-Monotonic)*: set operations
  - Attribute renaming **δ** (*Monotonic*)
- *Note: no duplicates or null*

- Projection **π**
  - Eliminate some columns → Keep all elements in X
  - R table name; X are attributes of R
  - $\boldsymbol{\pi_x(R)}$ [E.g. π_{title}(Schedule)]
  - Cut the table *vertically*

- Selection **σ**
  - Select tuples satisfying condition of the form (att =/≠ value/att_2)
  - $\boldsymbol{\sigma_{cond}(R)}$, where cond is a condition involving only attributes of R
  - Cut the table *horizontally*

- Union, Difference
  - Apply to tables with the same attributes
  - Union and difference of sets of tuples in R and S
  - Note: r - s: only keep those in R but not in S. (*p. 73, Feb 12)*

- Join
  - Operation combining tuples from two tables. (same as natural joins)
    - If common: Join on the matching terms

    - If no common: cross product

- Renaming
    - Change the name of attribute (can rename multiple at once)
    - The contents are not changed

- **Examples**

    - E.g. titles of currently playing movies
      π_{title}**(Schedule)**

    - E.g. titles of movies by Berto
      $\pi_{title}(\sigma_{director='Berto'}$**(Movie))**

    - E.g. titles and directors of currently playing movies
      π_{title}**(movie** $\bowtie$ **schedule)**

    - E.g. Pairs of actors acting together in some movie
      $\pi_{actor1,\ actor2}(\delta_{actor\ ->\ actor\ 1}$ **movie** $\bowtie$ $\delta_{actor\ ->\ actor\ 2}$**movie)**
        - Use join to match common terms
        - Join won't force matching on actors
        - Renaming helps use multiple variables
        - *Unique director assumption*

    - E.g. Find actors playing in every movie by Berto
        - Non-monotonic operations as difference
        - First, finds *all combination* of all actors and titles by Berto(cross product).
          π_{actor}**(movie)** $\bowtie$ $\pi_{titles}(\sigma_{director='Berto'}$**(Movie))**
        - Second, find actors for which there is a movie by Berto in which they do not act
          π_{actor}**(movie)** $\bowtie$ $\pi_{titles}(\sigma_{director='Berto'}$**(Movie))** - $\pi_{actor,title}$**(movie)**
          Actor title by Berto actors acts in title
          <r, t> belongs to the left side but no in the right hand side
        - Third, take complement of the above algebra.
          π_{actor}**(movie) -**
          π_{actor}**[(**π_{actor}**(movie)** $\bowtie$ $\pi_{titles}(\sigma_{director='Berto'}$**(Movie))** - $\pi_{actor,title}$**(movie)]**

- Other useful operations
    - Insertection ∩: can be changed into union and difference
    - Division /:
        - R / S : {a|<a,b> in R for **every b** in S}
        - Universal quantification
    - E.g. Find actors playing in every movie by Berto

$\pi_{titles,\ actor}(Movie) / \pi_{titles}(\sigma_{director='Berto'}(Movie))$

- Note
  - **π** is like “there exists” [existential quantification]
  - **Division /** is like “for all” [universal quantification]
  - R / S = $\pi_A(R) - \pi_A[(\pi_A(R) \bowtie S) - R]$
    <a> Find <a, b> that’s not in R

- Theorem
  - Calculus and algebra are **equivalent** (*p.84, Feb 12/14)*
    - Projection = existential quantification
    - Condition = t(A) compare c
    - Union = disjunction
    - Join = conjunction
    - Different = negation
    - Division = universal quantification

LEC 12 - Query Processing Cont. *Feb 14*

- Conversion between SQL, Calculus, Algebra

  - E.g. Find the theaters showing movies by Bertolucci
    - SQL

```
SELECT s.theater
FROM schedule s, movie m
WHERE s.title = m.title AND m.director = 'Berto"
```

    - Tuple Calculus
      {t: theater | ∃ s ∈ schedule ∃ m ∈ movie [t(theater) = m(theater) ⋀ s(title) = m(title) ⋀ m(director) = Berto] }
    - Relational algebra
      $\pi_{theater}(Schedule \bowtie \sigma_{director='Berto'}(movie))$

  - Note: number of items in **FROM clause = (number of joins + 1)**. Should minimize the number of joins, b/c joins are the most expensive

  - E.g. Find the drinkers who frequent some bars serving Coors
    - Relational algebra
      $\pi_{drinker}(frequent \bowtie \sigma_{beer='Coop'}(serves))$

  - E.g. Find the drinkers who frequent at least one bar serving a beer they like
    - Relational algebra
      $\pi_{drinker}(frequent \bowtie serves \bowtie likes)$

- E.g. Find the drinkers who frequent ONLY bars serving beer they like
  - Relational algebra
    $\pi_{drinker}$(frequent) - $\pi_{drinker}$[frequent - $\pi_{drinker, bar}$(serves ⋈ likes)]
    Drinker frequent a bar that doesn't serve a beer they like

- **Query Rewriting**
  Modify the query to make it more efficient and correct, without considering the current database.

  - Static Optimization
    - $R \bowtie R \rightarrow R$
    - $R - R \rightarrow \varnothing$
    - $\varnothing \bowtie R \rightarrow \varnothing$
    - R:AB, S: BC
    - Pushing selection: $\sigma_{A=a}(R \bowtie S) \rightarrow \sigma_{A=a}(R) \bowtie S$
    - Column eleminiaton: $\pi_{AB}(R \bowtie S) \rightarrow R \bowtie \pi_{B}(S)$
    - Cascading projections: $\pi_{A}(\pi_{A,B}(T)) \rightarrow \pi_{A}(T)$

  - Other Techniques
    - Nested (sub) query decorrelation
      - E.g. P(A, B), Q(C, D)

```
SELECT count(*) FROM p
WHERE a < (SELECT MAX(q.c) FROM q WHERE p.b = q.d
```

      - Rewrote

```
SELECT count(distinct p.*) FROM  p, q
WHERE b = d AND a < c
```

    - View unfolding
      - E.g. R(A, B, C)

```
CREATE VIEW v(d, e, f) AS SELECT a, b, sum(c) FROM
R
GROUP BY a, b;
SELECT d, sum(f) AS total FROM v GROUP BY D
```

      - Rewrote

```
SELECT a AS d, sum(C) AS total
FROM R
GROUP BY A
```

    - Common subqueries, etc.
    - Note: when duplicates are in the query (*p.97, Feb 14*)
      - E.g. R(A, B)

```
SELECT t.A FROM
(SELECT A, B FROM R GROUP BY A, B) AS t
```

- We **can** eliminate B from the SELECT clause.
- We **can't** eliminate B from the GROUP BY clause.

- **Query Evaluation Plan**

  - Additional decisions on evaluation of rewritten query, with partial information about database (Not static)
    - Statistical information on data
      - E.g. join relative small relations, selectivity (number of tuples by that is small)
    - Common subexpressions
    - Availability of indexes
      - E.g. data structure such as search tree. Auxiliary file providing fast access to physical location of record with given key value. If tree is balanced, access time is $\log_p(n)$ for p-ary trees. B^+ trees are always balanced.
      - Indexes allow specification of indexes, and query processors can create the index. And machine learning. (Database tuning)

  - More details on this (*p 102, Feb 14)*
  - Implementation of individual operations
    - Related to access method and file organization
      - Joining **unordered files takes $O(n^2)$ but ordered by some key field is O(nlog(n)**
    - Ordered file (sequential by name)

- Journey of a query
  - SQL → **Parser** → **Logical Plan Generator Applies Algebraic Rewriting** → **Logical Plan Generator** → Physical Plan Generators Chooses Execution Primitives And Data Passing → Physical Plan Generator

LEC 13 - Joins Minimization (*Feb 19)*

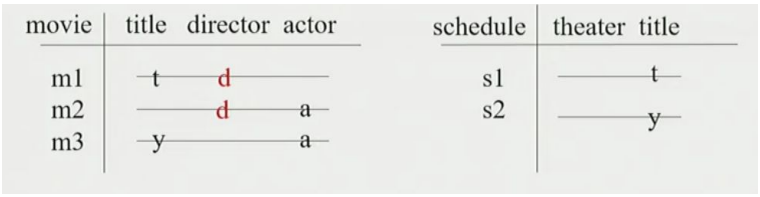
- Goal: minimize the number of tuples in the FROM clause (join minimization)

- E.g. Movie database:
  - SQL

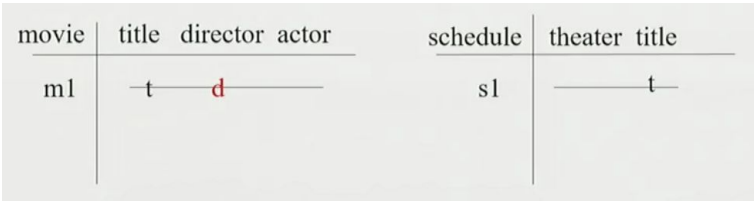
```
SELECT m1.director
```

```
FROM movie m1, movie m2, movie m3, schedule s1, schedule s1,
schedule s2
WHERE m1.director = m2.director AND m2.actor = m3.actor AND
m1.title = s1.title AND m3.title = s2.title
```

- Original



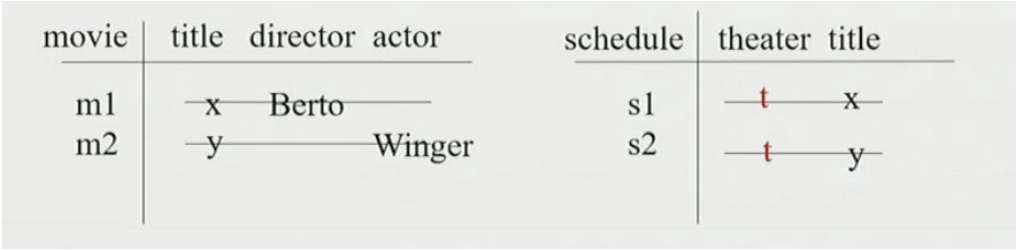
- Simplify to be



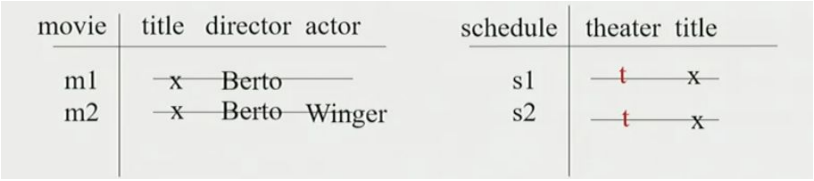
- E.g. Find theaters showing a title by Berto and a title in which Winger acts
  - SQL

```
SELECT s1.theater
FROM schedule s1, schedule s2, movie m1, movie m2
WHERE s1.theater = s2.theater AND s1.title = m1.title AND
m1.director = 'Berto' AND s2.title = m2.title AND m2.actor =
'Winger'
```

  - Original



  - This **cannot** be simplified.
    - But with additional constraints: each title has only one director and each theater shows only one title → x = y, m2.director = ‘Berto’



- Origins of Redundant Joins

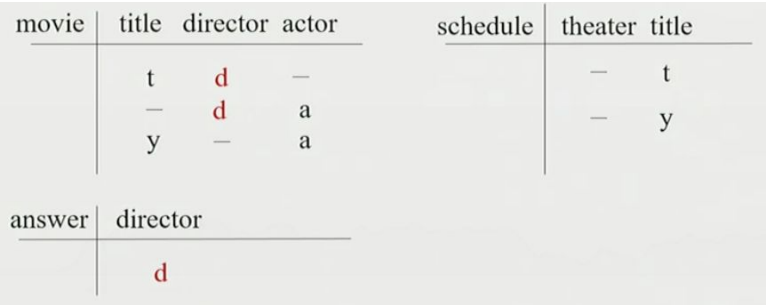
- Complex queries written by humans
- Queries resulting from view unfolding
    - E.g. Patience and doctors from “Scripps”, *(p. 85-96, Feb 19)*
- Very complex SQL queries generated by tools

- **Minimization Algorithm for Conjunctive SQL queries**

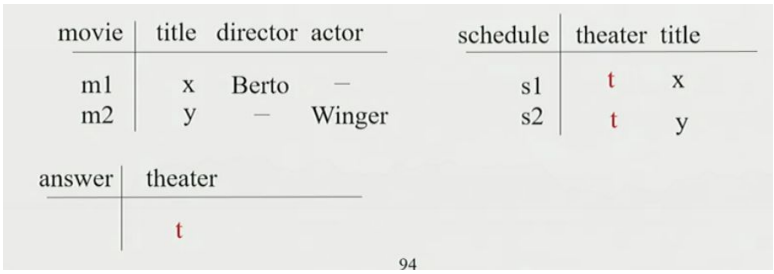
    - Basic Idea

        - Applied to SQL query whose **where** clause is a conjunction of equalities
        - Undecidable for simple SQL queries (halting problem for Turing machines)

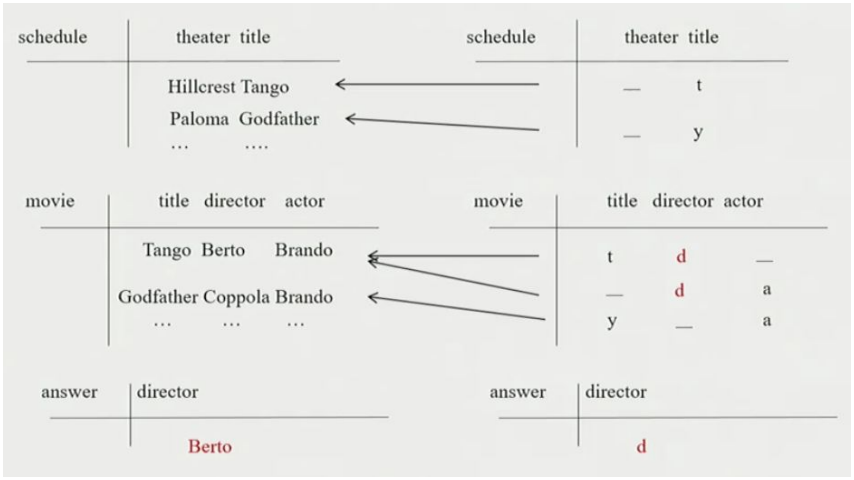
    - QBE patterns
        - Differences
            - No underscore to mark variables
            - Wildcards are explicitly denoted by “-” instead of blank
            - No insert.I in the answer relation

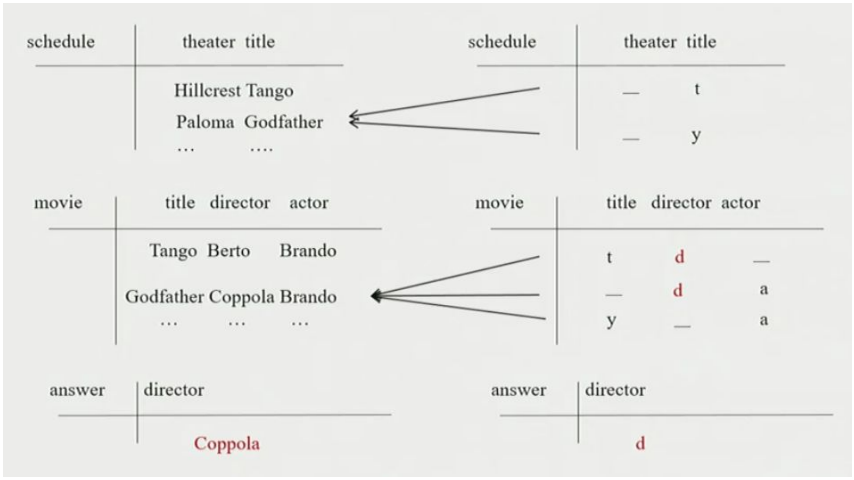


        - Convert SQL conjunctive queries to QBE patterns
            - E.g. Find theaters showing a title by Berto and a title in which Winger acts

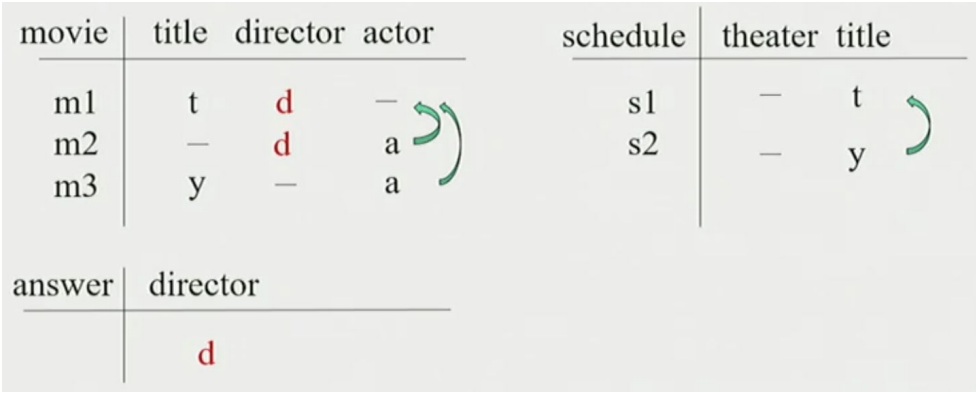


- Query defined by a pattern
  - **Matching** the pattern into the database





- Pattern Minimization



- Map the redundant rows to other rows. m1, s1 can play the role of m2, m3, s2.

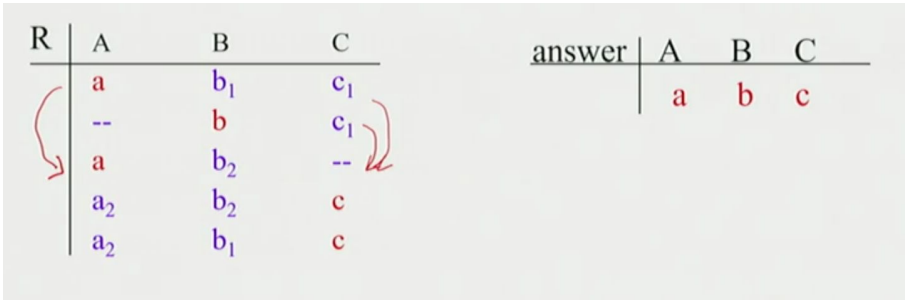
- **Pattern folding** (homomorphism)
  - Mapping **f** on variables, constants, wildcards of pattern **P** such that:
    - **f(x) = x** for **answer variable** and **constants x**. The answer variable and constant cannot be changed. Every other variable can be mapped anywhere (variable, wildcard, constant).
    - Every row of P is mapped to an **existing** row of P in the same relation. Do not invent new rows.

  - Theorem: if P is a pattern and f is a folding of P, then f(P) defines the same query as P.
    - Proof idea: answer of P on R = answer of f(P) on R
      - Double inclusion

LEC 14 - Continued (*Feb 21)*

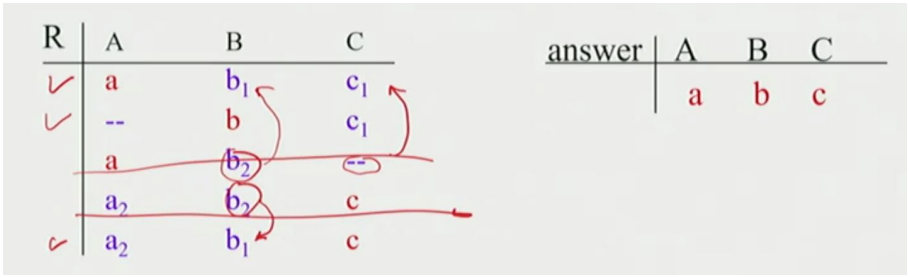
- Algorithm:
  - Repeatedly eliminate redundant rows **t is redundant if there is a folding f of P such that t doesn't belong to f(P)**.

- E.g. Global mapping
  All the **constraints** have to be satisfied by **all** tuples.
  - First row cannot be eliminated by mapping to row 3.



- If so, b_1 mapped to b_2 (all rows of b_1 mapped to b_2), and c_1 mapped to wildcard → b mapped to b_2. WRONG.

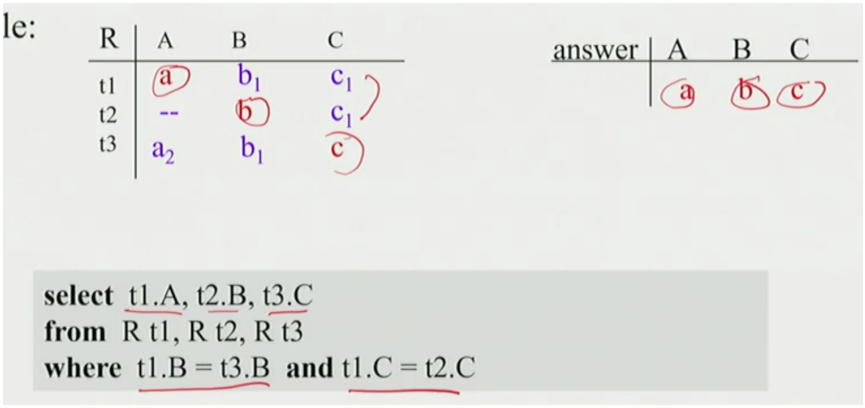
- Second row is the only with b, so cannot be mapped.
- Third row can be eliminated by mapping to row 5.



Wildcards and non-answer variables can map to anything else, but the mappings have to be consistent. (If you choose a mapping a → b, make sure it applies to everywhere)

- From Minimized Pattern to SQL Query

  - E.g. Previous example:



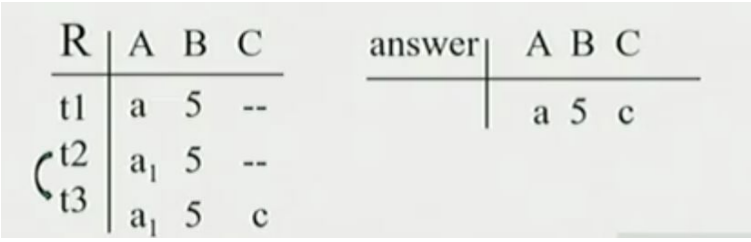
  - The answer is not unique, but the elements in the FROM clause are the same.

- A Complete Example

- SQL conjunctive query

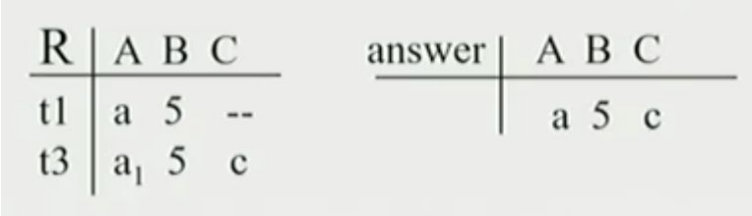
```
SELECT t1.a, t2.b, t3.c
FROM R t1, R t2, R t3
WHERE t2.a = t3.a AND t1.b = 5 AND t2.b = t3.b AND t2.b =
t1.b
```

- Pattern



  - Can map t2 to t3 but not t3 to t2.

- Minimized Pattern



- Minimized SQL query

```
SELECT t1.a, 5 as B, t3.c
FROM R t1, R t3
WHERE t1.b = 5 AND t3.b = 5
```

- Theorem: the minimization algorithm produces an SQL query with **minimum number of joins** among all conjunctive SQL queries equivalent to the original one on all databases.
  - Works better by extending with **functional dependencies**

- **Data Dependencies** (constraints)

  - Statements about valid data
    - Keys
    - Referential Integrity

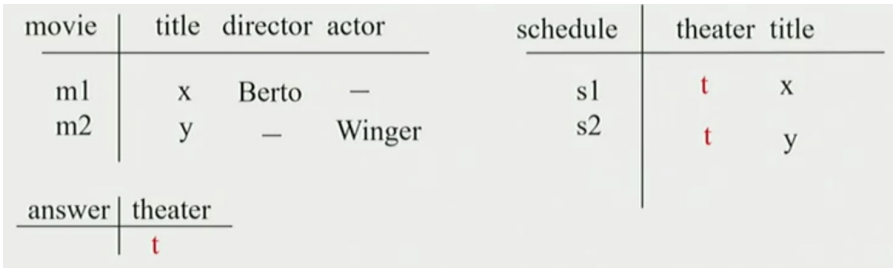
    - Functional dependencies: extension of keys
        - E.g. “Each employee works in no more than one department”
        - Name → Department
- Uses
    - Check data integrity
    - Query information
    - Schema Design → “Normal forms”

- Generalization
    - Some attributes may determine other attributes without being keys

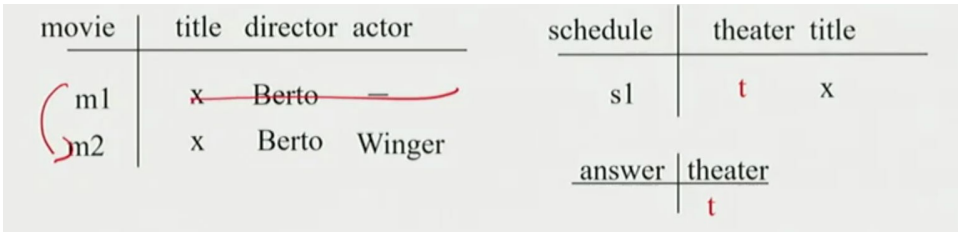
- **Functional Dependencies**
    - Expression X → Y where X, Y are attributes
    - An instance of R satisfies X → Y iff whenever two tuples agree on X, they also agree on Y.
    - Note: AB → CD are equivalent to AB → C and AB → D

- **CHASE**

    - E.g. Find theaters showing a title by Berto and a title in which Winger actors.



      The original pattern is minimal, when no FD assumed.
        - Suppose **title** → **director (**x = y then Berto = -), **theater** → **title** (x = y)

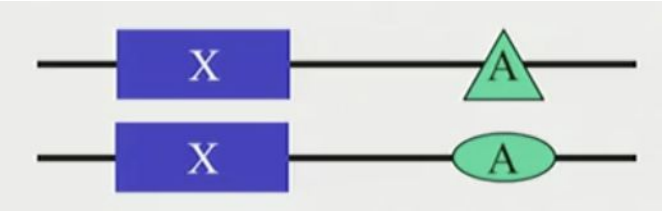


            - Force the pattern to satisfies the function dependence by changing the pattern itself.

    - Algorithm:
      Chasing the functional dependencies
        - Input: pattern P, a set F of FDs

- Output: pattern $CHASE_f(P)$ equivalent to P on all relations satisfying F
- Intuition: the chase modifies P so that it satisfies all FDs in F.

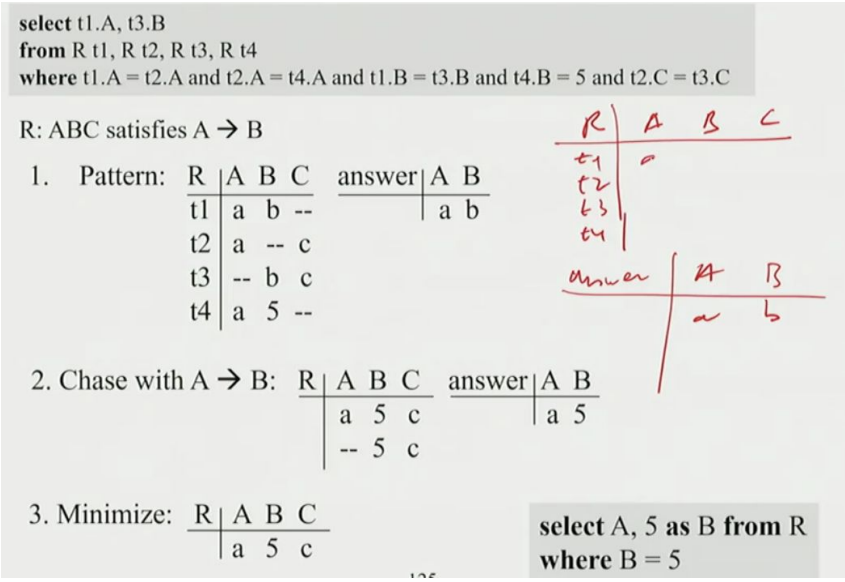
- Basic Step: $X \rightarrow A$
  - Eliminate the errors: if pattern contains two rows that agree on X and disagree on A, change them so that they also agree on A.



    - A, a_1: change a_1 to a.
    - A, wildcard: change wildcard to a
    - A, constant “Berto”: change a constant
    - Wildcard, wildcard: change both to *new* variable c_1, c_1
    - Two constants: remove two rows from the database: answer is empty

- **Optimization of SQL Conjunctive Queries with FDs**
  - Input: SQL conjunctive query Q, set of FDs F
    - Build pattern P of Q
    - Compute $CHASE_F(P)$
    - Minimize $CHASE_F(P)$
    - Construct the SQL query corresponding to the minimal pattern
  - Claim: **minimum possible number of joins** of SQL query equivalent to Q on all databases satisfying F.

  - Examples:



LEC 15 - Relational Database Design *(Feb 26)*

- Structure the information we want to store in the database:
  - Find the functional dependencies among the data (semantic assumptions about data)

- BAD relational database
  `BAD[#S, SNAME, SCITY, P#, PNAME, PCITY, QTY]`
  - Redundancy: S ~ SCITY appears many time
  - Update anomalies
    - Insertion: insert supplier before it supplies a part
    - Deletion: delete a part delete a supplier

- Normal forms: “nice forms” for schemas. *BCNF*
  New schema: no other functional dependencies besides key dependencies
  `S[S#, SNAME, SCITY]`
  `P[P#, PNAME, PCITY]`
  `SP[#S, #P, QTY]`

- Decomposition
  - Group the attributes into different tables such that original information is not lost
  - A **decomposition** of R is a set $\{R_1, R_2, \ldots, R_k\}$ of relation schema such that:
    $$\mathbf{att(R) = U_{i=1}^{k}\ att(R_i)}$$

- **Conditions for a Reasonable Decomposition**

  - **Lossless Join** Property [*Priority*]
    - The decomposition won't cause loss of information
      BAD $\subseteq$ π_s(BAD) $\bowtie$ π_p(BAD) $\bowtie$ π_{sp}(BAD)
    - The result of joining all projections is the combination of them
      - All tuples Vs. Correct tuples. All tuples doesn't mean more information.
    - The converse may hold, given the functional dependencies.

  - **Dependencies Preserving** Property
    - Dependencies for the original tables must be enforced by "local" dependencies on S, P, SP
    - Desirable for performance, efficient but not necessary

- **Check Lossless Join**

  - Find the query on the join of decomposed relations (conjunctive query)
    - E.g. the pattern of selecting from three decompositions

S#	SNAME	SCITY	P#	PNAME	PCITY	QTY
a_1	a_2	a_3	--	--	--	--
--	--	--	a_4	a_5	a_6	--
a_1	--	--	a_4	--	--	a_7
a_1	a_2	a_3	a_4	a_5	a_6	a_7

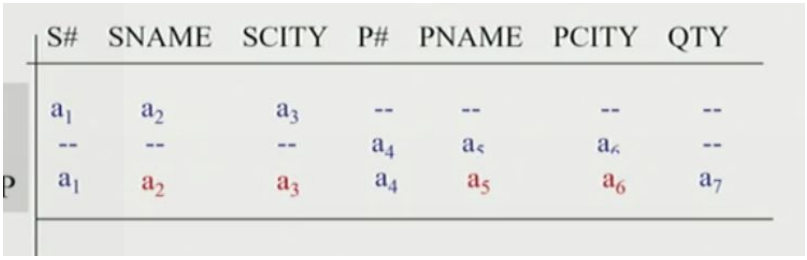
  - Run the join minimization CHASE algorithm
    - E.g. based on function dependencies {S# → SCITY NAME, P# → PNAME PCITY, P#C# → QTY}

S#	SNAME	SCITY	P#	PNAME	PCITY	QTY
a_1	a_2	a_3	--	--	--	--
--	--	--	a_4	a_5	a_6	--
a_1	a_2	a_3	a_4	a_5	a_6	a_7
a_1	a_2	a_3	a_4	a_5	a_6	a_7

  - Check the output against the query on the relations:
    - find a row of answer variables

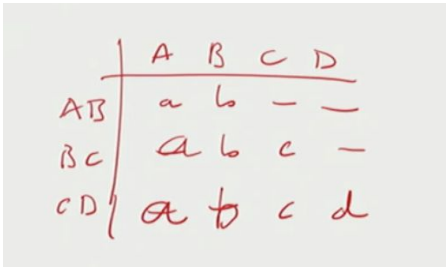
  - **More concise algorithm**
    - Construct a pattern using variable a for each attribute A. For each R_i the pattern has one row with variable a for each attribute A of R_i and

wildcards everywhere else.
- Chase the pattern with F
- Output YES iff the resulting pattern has an entire row of variables
- E.g. previous example revisited:



S#	SNAME	SCITY	P#	PNAME	PCITY	QTY
a_1	a_2	a_3	--	--	--	--
--	--	--	a_4	a_5	a_6	--
a_1	a_2	a_3	a_4	a_5	a_6	a_7

- E.g. Decompose R={A,B,C,D} into AB, BC, CD, with F = {B → A, C → B}



- **Check Dependency Preservation**

  - Function Dependencies Implication

    - Definition: a set F of FDs **implies** another FD X → Y if every relation satisfying F also satisfies X → Y.
      - Notation: **F |= X → A**
      - Trivial: left hand side includes right hand side

    - Given FDs can imply additional FDs: every relation that satisfies A → B and B → C also satisfies A → C.
      - A → B and B → C **imply** A → C

    - Reachability question: Can I reach E from AB given the functional dependency.
      - A → C, BC → D, AD → E **imply** AB → E

  - **Closure of X**:

    - Definition: the closure of a set of attributes X with respect to a set F of FDs is **X^+ = {A | F |= X → A}**
      - The set of attributes "*determined*" by X

- Usage: compute keys, check dependency preservation, check satisfaction of normal forms

- Closure Computation: since att(R) is finite, there must exists $X^+ = X^{(k)}$
  Repeat until $X^{(i+1)} = X^{(i)}$
  - $X^{(0)} \leftarrow X$
  - $X^{(i+1)} \leftarrow X^{(i)} \cup \{Z \mid V \rightarrow Z \in F, V \subseteq X^{(i)}\}$
  - Then finally we get $X^{(0)} \subseteq X^{(1)} \subseteq \ldots. \subseteq att(R)$

- **Local FDs**:

  - If F is a set of FD's over R and X belongs to attributes(R) then
    $\pi_X(F^+) = \{V \rightarrow W \in F^+ \mid V \subseteq X \text{ and } W \subseteq X\}$
  - These are the FDs implied by F that apply to the set of attributes X (Local to X)
  - E.g. F = {A → C, BC → D, AD → E}
    - Local to AC: A → C
    - Local to ABD: AB → D
      - $A^+ = AC$, A → C not local to ABD, C not part of ABD
      - $B^+ = B$, $D^+ = D$
      - $AB^+ = \underline{AB}C\underline{D}E$, $AD^+ = \underline{AD}CE$, $BD^+ = BD$

- Dependency Preserving Decompositions

  - All FDs in F are implied by the local FDs
  - Let p = $\{R_1, R_2, \ldots, R_k\}$ be a decomposition for R and F a set of FD's over R, then p **preservese** F iff:
    The set of local FDs is equivalent to F.

- Naive Algorithm
  - Compute F^+
  - Compute G = all local FDs % local FDs
  - Check if F is a subset of G^+
  - *Note*: impractical since the size of F^+ can be exponential

LEC 15 - Continued *Feb 28*

- Improved Algorithm

  - Avoid computing all G: X → A is local to R_i iff AX is a subset of $att(R_i)$ and A is an element of X^+

- Get from X → Y using only local FDs. Keep adding local attributes based on FDs on start variable, and then keep adding the attributes to see if arrive at desired variable.
- E.g. R = {A, B, C, D}, P = {AB, BC, CD}, F = {A → B, B → C, C → D, D → A}; Does P preserves *D → A* ?
    - Decomposition: Start with D
        - D^+ = DABC so D → C is local to CD, add C
        - C^+ = CDBA so C → B is local to BC, add B
        - B^+ = BCDA so B → A is local to AB, add A, success.

- Pseudocode (pushing selection)

```
For each X → F in F do
  Z := X
  while changes occur in Z do:
  For i := 1 to k do
    Z := Z union ((Z intersect R_i)^+ intersect R_i)
  If Y is not in Z output "no" and stop
Output "yes"
```

- **Normal Forms**

    - Recall:
        - Superkey: X determines all attributes of R
        - Key: X is a minimal superkey
        - *Note: if an attribute never appears on the right hand side of any functional dependency, all key must contain this attribute.*

    - Purpose: eliminate problems of redundancy and anomalies

    - **Boyce-Codd Normal Form** (BCNF)

        - Definition: a relation schema R is in BCNF wrt a set of FD's F over R, iff whenever X → A is an element of F^+ andA is not an element of X (non-trivial), **X is a superkey for R**.
            - An individual table property

        - A relation is BCNF if its functional dependencies are consequences of **keys**. In other words, the left hand side of its functional dependencies are keys.
            - Every key must contain the attributes that never appears in the right-hand side.

- Every **two-attribute** relation is itself a BCNF.

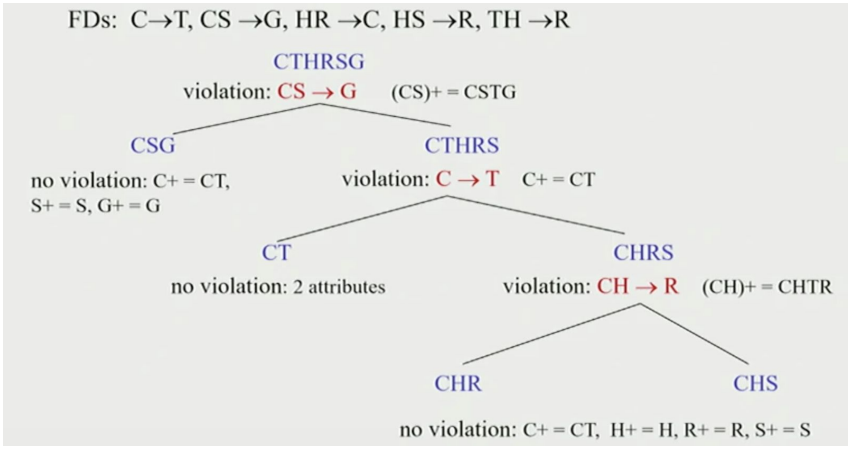
- Decomposition
  - Any relation schema has a decomposition into BCNF relation schemas, with *lossless join*, but **not** always *dependency preservation*.

- Algorithm
  - Basic Step: eliminate the violation of BCNF while checking the lossless join.
    - E.g. S = ABC with local FD A → B
      - Violates BCNF b/c A is not a superkey of S
      - Then divide S into AB and AC.
      - Check lossless join
  - General Approach:

```
Start with p = {R}
Apply recursively the following procedure:

     For each S in p not in BCNF
         Let X → A element of F+ be such that XA
subset of S, A not element of X, X not a superkey of
S
      Replace S by S1 and S2 where
            S1 = XA
            S2 = X(S - A)
Until all relation schemes are in BCNF
```

- Example
  - First, find all attributes that doesn't belong to the right hand side. Then all keys must contain them.
  - Second, randomly pick any violation and resolve it.



- Non-deterministic: different decompositions
- Lossless join, but not dependency preserving
    - E.g. Doesn't preserve TH → R
- *Note:* checking if a relation is BCNP is NP-complete; not decomposed to get **both** lossless join **and** preserves the dependencies

(March 5th)

- **Third Normal Form** (3NF)

    - Provides some trade off, satisfy both lossless join and preserves the dependencies. 3NF is weaker than BCNF.

    - Definition: a relation scheme R is in **third normal form** wrt a set F of FDs over F, if whenever X → A holds in R and A is not an element of X then either X is a superkey or A is **prime** (A is an element of K for some key K).

    - E.g. R = city, zip, street; F = city street → zip, zip → city
        - Not in BCNF: ZIP → CITY, ZIP is not a key to R, while ZIP, ST is.
        - In 3NF: city belongs to the key {CITY, STREET}

    - Algorithm
        - Simplify the set of FD (eliminate redundancies)
        - Construct a first cut decomposition from remaining FDs
        - If needed, modify decomposition to ensure lossless join

    - **Step1:** Eliminate redundancies in the FDs

- Rewrite FDs with single attributes on RHS
- Eliminate redundant FDs (those already implied by other FDs)
- Eliminate redundant attributes from LHS of FDs

- **Step2:** First-cut 3 NF decomposition
  - R a schema, F a minimal set of FDs over R (without redundancies): P = $\{XA_1A_2...A_m \mid X \rightarrow A_i \text{ in } F\}$ union other attributes not occur in F
  - *Theorem:* P preserves F and each R_i in P is in 3NF wrt $PI_{Ri}(F^+)$.
    - Suppose XA is constructed from $X \rightarrow A$ in F. Let $Y \rightarrow B$ be local to XA:
      - 1. B = A, then Y = X and Y is a superkey (if Y belongs to X then attributes in X - Y are redundant).
      - 2. $B \neq A$, so B is element of X so B is prime (X must be a key otherwise $X \rightarrow A$ has redundant attributes on left hand side).

- **Step3:** Second-cut ensure lossless join
  - Add p to a set K where **K is a key for R**, unless there already is a "piece" in decomposition whose attributes form a superkey for R.
  - *Theorem:* P union {K} is a 3NF which is dependency preserving and lossless join.
  - Test for Lossless join: chase in the order of FDs used in the computation of the superkey.

- **(Optional) Step4:** eliminate the subsets

- **Schema Design**

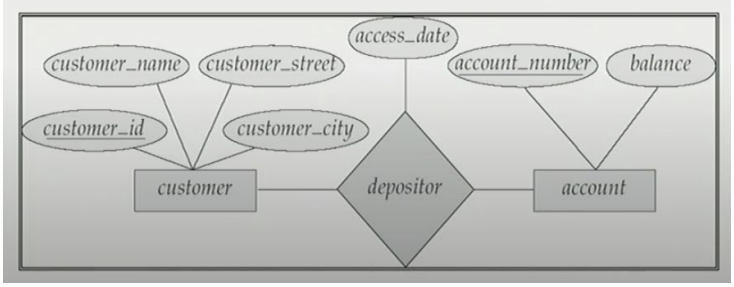
  - Overview
    - Choose attributes R
    - Specify dependencies F (Armstrong)
    - Find a lossless-join, dependency preserving decomposition into normal forms schemas:
      - BCNF, if possible
      - 3NF, if not BCNF

  - Attributes vs. Data
    - Employment: {name, department} Vs. {name, PC, toys, candy}
      - Find a department of Jack. → One
      - Find all employees in a department → Two (monotonic)

- Armstrong relation: satisfies exactly the FDs in F^+, violates anything that's not in F^+.

- A Glimpse Beyond FDs
  - MVD Dependencies:
    - State the dependency given some fixed attributes (lossless join).
    - E.g. in the movie database, the directors and actors are independent information for the same title. Then a better design is to put directors and actors in two separate tables.
    - Taking into account MVD results of BCNF as *4th normal form*. Algorithm is very complicated.
  - Foreign Keys (referential integrity constraints)
    - Computation is undecided (every foreign key in other relations is a key) → algorithmic impossible
    - Can't check the functional dependency is implied
    - Some restrictions can make these algorithms work (cycle, etc.)

- Alternative Approach
  - Start with an Entity-Relationship (ER) diagram. Translate into corresponding relational schema. Identify functional dependencies. Use design theory to further improve and bring to BCNF or 3NF or 4NF.
  - E.g. Bank database



LEC 17 - Concurrency Control (*March 7)*

- Underneath implementation level: concurrency control
  - Multiple users may access the database at the same time

- **Transaction**: execution of a program having the following **Properties**:
  - Atomicity: a transaction either happens or doesn't, either completes and the results become visible or no results are visible
  - Consistency: transactions preserve correctness of database
  - Isolation: each transaction is unaware of other transactions executing concurrently

- Durability: results of a completed transaction are permanently installed within D.B.
- **ACID** properties

- **Transaction**: has the two possible **Outcome**
  - Commit: Program execution completes and the results become permanent in the database
  - Abort: Program execution was not successful. “Results” are not installed into database

- **Guarantee serializability:** Main idea of Concurrency Control
  - An execution without any interleaving is OK
  - If an execution has the same effect as a serial execution then it’s also acceptable

- E.g. constraint: A = B and T1 and T2.

T1	**T2**
Read(A)	Read(A)
A ← A + 100	A ← A x 2
Write(A)	Write(A)
Read(B)	Read(B)
B ← B + 100	B ← B x 2
Write(B)	Write(B)

  - Serial schedule: T1 followed by T2 or T2 followed by T1
    - *Note: concurrency control doesn’t distinguish among different serialization*
  - Interleaved schedule:
    - C: Equivalent to serial schedule T1 followed by T2 because the same initial state leads to the same end state
    - D: Bad schedule because constraints doesn’t hold. (Can’t be serialized)
    - E: Correct by accident being semantics, still bad

- Test Serializability: only on the order of read/write schedule
  - E.g. SC = $r_1(A)w_1(A)....r_2(B)w_2(B)$

- **Conflict equivalent:** S_1 can be transformed into S_2 by a series of swaps of adjacent non-conflicting actions:
  - Actions on different data

- Read/read on the same data

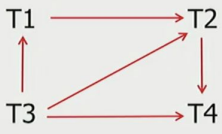
- **Conflict Serializable**: if the schedule is *conflict equivalent* to some serial schedule.

  - E.g. Swap the order
    SC = $r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$
    SC' = $r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$ → T_1 followed by T_2
    - Acyclic: T_1 always comes before T_2.

  - E.g. Not possible:
    SD = $r_1(A)w_1(A)$~~$r_4(B)w_4(B)$~~$r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B)$
    - More general: $T_1 \rightarrow T_2$ and T2 → T_1: circle and SD cannot be rearranged into serial schedule, SD is not "equivalent" to any serial schedule. SD is "bad"

- **Precedence Graph** P(S) [S as a schedule]
  - Nodes: transactions in S; Edge $T_i \rightarrow T_j$ whenever:
    - $P_i(A),Q_j(A)$ are actions in S
    - $P_i(A) <_s P_j(A)$
    - At least one of P_i, Q_j is a write
  - E.g. Find the P(S) for the following:

    S = $w_3(A)\ w_2(C)\ r_1(A)\ w_1(B)\ r_1(C)\ w_2(A)\ r_4(A)\ w_4(D)$

    - No edge between T_1 to T_4 because $r_1(A)$ and $r_4(A)$ are both read.
    - T_2 to T_1 because $w_2(C)$ and $r_1(C)$ → circle. **NOT** serializable.

- **Proof**
  - Lemma: **S_1, S_2 conflict equivalent → $P(S_1) = P(S_2)$**
    Assume S_1 and S_2 are conflict equivalent and $P(S_1) \neq P(S_2)$. Then there exits $T_i \rightarrow T_j$ in S_1 and not in S_2. Thus $S_1 = ..p_i(A)..q_j(A)...$ and $S_1 = q_j(A)...p_i(A)...$ Then p_i, q_j is conflict and S_1 and S_2 are not conflict equivalent. **Contradiction.**

  - Note: $P(S_1) = P(S_2)$ doesn't imply S_1, S_2 are conflict equivalent.
    Counter example: $S_1 = w_1(A)r_2(A)w_2(B)r_1(B)$; $S_2=r_2(A)w_1(A)r_1(B)w_2(B)$. Both are $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$. But they are not conflict equivalent.

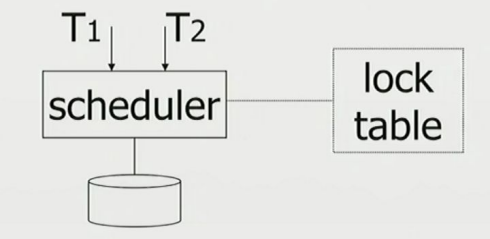
  - Theorem: **$P(S_1)$ ⇐⇒ S_1 conflict serializable**

- Assume S_1 is conflict serializable, there exists an S_s such that S_s and S_1 are conflict equivalent. Then $P(S_s) = P(S_1)$ and therefore $P(S_1)$ is acyclic since $P(S_s)$ is acyclic.
- Assume $P(S_1)$ is acyclic, then move all T_k that has no incoming arcs to the front recursively. And finally we can serialize the entire schedule. $S_1 = <T_k$ action><..rest..>.
- Any topological sort of the precedence graph is serializable.

- **Enforce Serializable Schedules**

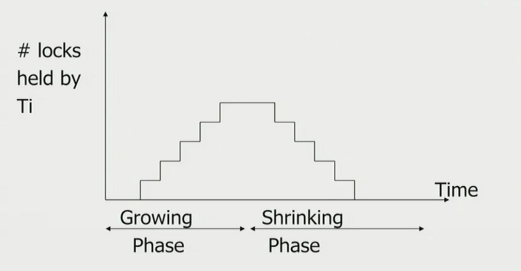
  - *Optimistic Concurrency Control*: run system, recording P(S) check for P(S) cycles and declare if execution was good or abort transactions as soon as they generate a cycle.
    - Cycle not occuring often

  - *Option 2* (More PRACTICAL): Prevent P(S) cycles from occurring



- **Lock**
  - A locking protocol:
    - Lock (exclusive): $l_i(A) \rightarrow T_i$ locks on A
    - Unlock: $u_i(A) \rightarrow T_i$ unlocks on A
  - Rule 1: Well-formed Transactions: in order for a transaction to access an entity, it must lock it.
    $T_i = ...l_i(A) ...p_i(A) \ldots u_i(A) \ldots$
  - Rule 2: Legal Scheduler: only one transaction that can have a lock.
    $S = \ldots l_i(A) \ldots u_i(A)...$ with no $l_j(A)$ in between.

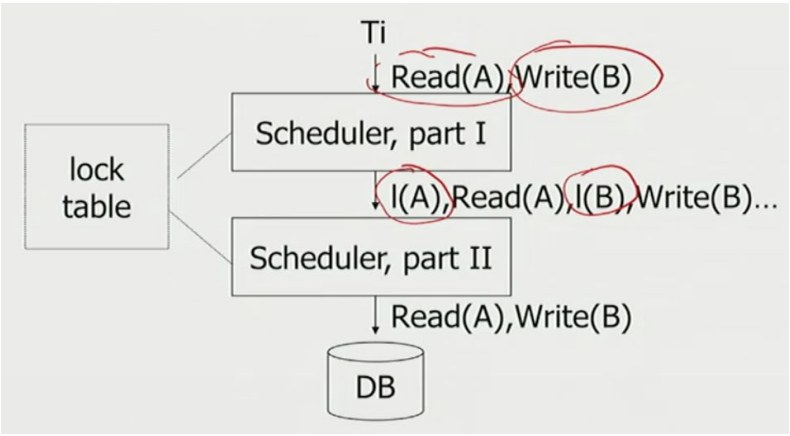
- **Two Phase Locking** (2PL)
  - Transaction has two phases: locking phase and unlocking phase.
    $T_i = \ldots l_i(A) \ldots. U_i (A) \ldots.$



- May result in deadlock, then abort one of the transactions.

- Theorem 2PL ⇒ conflict serialization
  - Definition: shrink(T_i) = SH(T_i) = first unlock action of T_i
  - Lemma: $T_i \rightarrow T_j$ in S ⇒ SH(T_i) $<_s$ SH(T_j).
  - Proof: Assume P(S) has cycle, by lemma SH(T_1) < SH(T_2) < .. SH(T_1), impossible so P(S) is acyclic and S is conflict serializable.

- Sample Locking System
  - Don't trust transactions to request / release locks
  - Hold all locks until transaction commits.



- The objects to lock
  - Relations/Tuple/Disk block
  - Large objects (e.g. Relations): need few locks and low concurrency
  - Small objects (e.g. tuples, fields): need more locks and more concurrency

Practice Problems

Recursion

- Question 1: Find the pairs of stations (a, b) such that b can be reached by some combination of metro and bus, but not the metro or the bus alone.
  - Find combination of routes

```
create recursive view Combination as
(select * from Metro) union (select * from Bus)
union
select a.station, b.next_station
from combination a, combination b
where a.next_station = b.station
```

  - Find routes by bus/metro alone

```
Create recursive view MT as
(select * from Metro)
Union
Select Metro.station, MT.next_station
From Metro, MT
Where Metro.next_station = MT.station
```

  - Find the set difference

```
Select * from combination
Except
(select * from MT) union (select * from BT)
```

- Question 2: Find all trusted nodes in Flow

```
CREATE RECURSIVE VIEW Trusted as
      SELECT n.id FROM node n
      WHERE NOT EXISTS
            (SELECT * FROM Flow
             WHERE Flow.to = n.id AND Flow.from
                                  NOT IN (SELECT * FROM Trusted))
```

- Question 3: Find all before id1, id2 in the binary tree defined by Left(parent, child), Right (parent, child)
  - Find all edges in the binary tree

```
CREATE VIEW Edge AS
SELECT parent AS id1, child AS id2 FROM left
UNION
```

```
SELECT parent as id1, child as id2 FROM right
```

- Find all descendent relationships

```
CREATE RECURSIVE VIEW Descendant AS
SELECT * FROM Edge
UNION
SELECT e.id1, d.id2 FROM Edge e, Descendant d
WHERE e.id2 = d.id1
```

- Combine all “descendant relationships”, “left and right relationship” and “left and right in the descendent relationships”

```
SELECT * FROM Descendant
UNION
SELECT l.child as id1, r.child as id2
FROM left l, right r
WHERE l.parent = r.parent
UNION
SELECT x.id2 as id1, y.id2
FROM left l, right r, Descendant x, Descendant y
WHERE l.parent = r.parent AND x.id1 = l.child AND y.id1 =
r.child
```

- Gradiance:

```
5) WITH RECURSIVE Ancestor(X,Y) AS
     ( (SELECT par,ch FROM Parent WHERE par='Eve')
        UNION
       (SELECT A1.X, A2.Y
        FROM Ancestor A1, Ancestor A2
        WHERE A1.Y = A2.X) )
```

  - This doesn’t work for either A and B. Stuck with first recursive iteration.

Relational Algebra

- List actors cast only in movies directed by Berto
  - Find titles not by Berto
  - Find actors performs in the titles not by Berto
  - Difference all actors with actors perform in the titles not by Berto

- List directories such that every actor is cast in one of his/her movies
  - Division, or
  - Find all combinations of director and actors

    - Find actors and directors that actually work together
    - Take difference of above two to find directors who hasn't collaborated with some actors
    - Then take a difference of all directors and those who didn't collaborated with some actors

- Proof of distributivity
- Rewrite query by minimizing the size of tables being joined. (Pushing conditions inside, etc.)

Join Minimization

Functional Dependencies

- Find keys by first checking the things not in right hand side
- Determine the super keys
- Compute F^+

Normal Forms

- Check BCNF
- Check 3NF
- Check lossless join
- Check functional dependency

Concurrency Control

- Precedence graph from given serial transaction
- Two phase locking and serializability
- Concurrency control protocol: serializability and deadlocks
  No 2 transactions on same database
- **2. A transaction cannot read or write a data item (X or Y) without holding the lock on that item; i.e., the lock action occurs before and the unlock occurs after the read or write.**
- **3. Both transactions cannot hold a lock on the same data item at the same time. That is, the second can only lock an item after the first unlocks it.**
- **No transaction may lock an item after it has unlocked another item (the two-phase-locking condition).**