Data exchange

- Source schema, target schema; need to transfer data between them.
- A typical scenario:
  - Two organizations have their legacy databases, schemas cannot be changed.
  - Data from one organization 1 needs to be transferred to data from organization 2.
  - Queries need to be answered against the transferred data.
Data Exchange

![Data Exchange Diagram]

Source Schema \( S \)  
Target Schema \( T \)
Data Exchange

Source Schema $S$  

Source Database

Target Schema $T$

Target Database

L. Libkin

Data Integration and Exchange
Data exchange: an example

• We want to create a target database with the schema

\[ \text{Flight(city1, city2, aircraft, departure, arrival)} \]
\[ \text{Served(city, country, population, agency)} \]

• We don’t start from scratch: there is a source database containing relations

\[ \text{Route(source, destination, , departure)} \]
\[ \text{BG(country, city)} \]

• We want to transfer data from the source to the target.
Data exchange – relationships between the source and the target

How to specify the relationship?
Relationships between the source and the target

• Formal specification: we have a *relational calculus query* over both the source and the target schema.

• The query is of a restricted form, and can be thought of as a sequence of rules:

\[
\text{Flight}(c_1, c_2, __, \text{dept}, __) \leftarrow \text{Route}(c_1, c_2, \text{dept})
\]

\[
\text{Served}(\text{city}, \text{country}, __, __) \leftarrow \text{Route}(\text{city}, __, __), \text{BG}(\text{city}, \text{country})
\]

\[
\text{Served}(\text{city}, \text{country}, __, __) \leftarrow \text{Route}(__, \text{city}, __), \text{BG}(\text{city}, \text{country})
\]
Data exchange – targets

- Target instances should satisfy the rules.
- What does it mean to satisfy a rule?
- Formally, if we take:

  \[ \text{Flight}(c_1, c_2, \_, \text{dept}, \_) \leftarrow \text{Route}(c_1, c_2, \text{dept}) \]

  then it is satisfied by a source \( S \) and a target \( T \) if the constraint

  \[ \forall c_1, c_2, d \left( \text{Route}(c_1, c_2, d) \rightarrow \exists a_1, a_2 \left( \text{Flight}(c_1, c_2, a_1, d, a_2) \right) \right) \]

- This constraint is a relational calculus query that evaluates to \textit{true} or \textit{false}
Data exchange – targets

• What happens if there no values for some attributes, e.g. *aircraft*, *arrival*?

• We put in *null values* or some real values.

• But then we may have multiple solutions!
Data exchange – targets

Source Database:

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Departure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edinburgh</td>
<td>Amsterdam</td>
<td>0600</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>London</td>
<td>0615</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>Frankfurt</td>
<td>0700</td>
</tr>
</tbody>
</table>

BG:

<table>
<thead>
<tr>
<th>Country</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>UK</td>
<td>London</td>
</tr>
<tr>
<td>UK</td>
<td>Edinburgh</td>
</tr>
<tr>
<td>NL</td>
<td>Amsterdam</td>
</tr>
<tr>
<td>GER</td>
<td>Frankfurt</td>
</tr>
</tbody>
</table>

Look at the rule

\[ \text{Flight}(c1, c2, _, \text{dept}, _) :\text{ Route}(c1, c2, \text{dept}) \]

The right hand side is satisfied by

\[ \text{Route}(\text{Edinburgh}, \text{Amsterdam}, 0600) \]

But what can we put in the target?
Data exchange – targets

Rule: \( \text{Flight}(c1, c2, \_ , \_ , \text{dept}, \_ ) \leftarrow \text{Route}(c1, c2, \text{dept}) \)

Satisfied by: \( \text{Route}(\text{Edinburgh, Amsterdam, 0600}) \)

Possible targets:

- \( \text{Flight}(\text{Edinburgh, Amsterdam, } \bot_1, 0600, \bot_2) \)
- \( \text{Flight}(\text{Edinburgh, Amsterdam, B737, 0600, } \bot) \)
- \( \text{Flight}(\text{Edinburgh, Amsterdam, } \bot, 0600, 0845) \)
- \( \text{Flight}(\text{Edinburgh, Amsterdam, } \bot, 0600, \bot) \)
- \( \text{Flight}(\text{Edinburgh, Amsterdam, B737, 0600, 0845}) \)

They all satisfy the constraints!
Which target to choose

• One of them happens to be right:
  – Flight(Edinburgh, Amsterdam, B737, 0600, 0845)

• But in general we do not know this; it looks just as good as
  – Flight(Edinburgh, Amsterdam, ’The Spirit of St Louis’, 0600, 1300), or
  – Flight(Edinburgh, Amsterdam, F16, 0600, 0620).

• Goal: look for the “most general” solution.

• How to define “most general”: can be mapped into any other solution.

• It is not unique either, but the space of solution is greatly reduced.

• In our case Flight(Edinburgh, Amsterdam, $\bot_1$, 0600, $\bot_2$) is most general as it makes no additional assumptions about the nulls.
Universal solutions

- A homomorphism is a mapping $h : \text{Nulls} \rightarrow \text{Nulls} \cup \text{Constants}$.
- For example, $h(\perp_1) = B737$, $h(\perp_2) = 0845$.
- If we have two solutions $T_1$ and $T_2$, then $h$ is a homomorphism from $T_1$ into $T_2$ if for each tuple $t$ in $T_1$, the tuple $h(t)$ is in $T_2$.
- For example, if we have a tuple
  \[ t = \text{Flight}(\text{Edinburgh, Amsterdam, } \perp_1, 0600, \perp_2) \]
  then
  \[ h(t) = \text{Flight}(\text{Edinburgh, Amsterdam, B737, 0600, 0845}). \]
- A solution is universal if there is a homomorphism from it into every other solution.
- (We shall revisit this definition later, to deal with nulls properly.)
Universal solutions: still too many of them

• Take any \( n > 0 \) and consider the solution with \( n \) tuples:

  \[
  \text{Flight(Edinburgh, Amsterdam, } \bot_1, \ 0600, \ \bot_2) \\
  \text{Flight(Edinburgh, Amsterdam, } \bot_3, \ 0600, \ \bot_4) \\
  \ldots \\
  \text{Flight(Edinburgh, Amsterdam, } \bot_{2n-1}, \ 0600, \ \bot_{2n})
  \]

• It is universal too: take a homomorphism

  \[
  h'(\bot_i) = \begin{cases} 
  \bot_1 & \text{if } i \text{ is odd} \\
  \bot_2 & \text{if } i \text{ is even}
  \end{cases}
  \]

• It sends this solution into

  \[
  \text{Flight(Edinburgh, Amsterdam, } \bot_1, \ 0600, \ \bot_2)
  \]
Universal solutions: cannot be distinguished by conjunctive queries

- There are queries that distinguish large and small universal solutions (e.g., does a relation have at least 2 tuples?)
- But these cannot be distinguished by conjunctive queries
- Because: if \( \bot_{i_1}, \ldots, \bot_{i_k} \) witness a conjunctive query, so do \( h(\bot_{i_1}), \ldots, h(\bot_{i_k}) \) — hence, one tuple suffices
- In general, if we have
  - a homomorphism \( h : T \rightarrow T' \),
  - a conjunctive query \( Q \)
  - a tuple \( t \) without nulls such that \( t \in Q(T) \)
- then \( t \in Q(T') \)
Universal solutions and conjunctive queries

• If
  ○ $T$ and $T'$ are two universal solutions
  ○ $Q$ is a conjunctive query, and
  ○ $t$ is a tuple without nulls,
then
  $$t \in Q(T) \iff t \in Q(T')$$
because we have homomorphisms $T \rightarrow T'$ and $T' \rightarrow T$.

• Furthermore, if
  ○ $T$ is a universal solution, and $T''$ is an arbitrary solution,
then
  $$t \in Q(T) \Rightarrow t \in Q(T'')$$
Universal solutions and conjunctive queries cont’d

• Now recall what we learned about answering conjunctive queries over databases with nulls:
  o $T$ is a naive table
  o the set of tuples without nulls in $Q(T)$ is precisely $\text{certain}(Q, T)$ – certain answers over $T$

• Hence if $T$ is an arbitrary universal solution

\[
\text{certain}(Q, T) = \bigcap \{Q(T') \mid T' \text{ is a solution}\}
\]

• $\bigcap \{Q(T') \mid T' \text{ is a solution}\}$ is the set of certain answers in data exchange under mapping $M$: $\text{certain}_M(Q, S)$. Thus

\[
\text{certain}_M(Q, S) = \text{certain}(Q, T)
\]

for every universal solution $T$ for $S$ under $M$. 
Universal solutions cont’d

- To answer conjunctive queries, one needs an arbitrary universal solution.
- We saw some; intuitively, it is better to have:

  \[
  \text{Flight}(\text{Edinburgh, Amsterdam, } \perp_1, 0600, \perp_2)
  \]

  than

  \[
  \begin{align*}
  \text{Flight}(\text{Edinburgh, Amsterdam, } &\perp_1, 0600, \perp_2) \\
  \text{Flight}(\text{Edinburgh, Amsterdam, } &\perp_3, 0600, \perp_4) \\
  \cdots \\
  \text{Flight}(\text{Edinburgh, Amsterdam, } &\perp_{2n-1}, 0600, \perp_{2n})
  \end{align*}
  \]

- We now define a **canonical** universal solution.
Canonical universal solution

- Convert each rule into a rule of the form:

  \[ \psi(x_1, \ldots, x_n, z_1, \ldots, z_k) \leftarrow \varphi(x_1, \ldots, x_n, y_1, \ldots, y_m) \]

  (for example,

  \[ \text{Flight}(c_1, c_2, _, \text{dept}, _) \leftarrow \text{Route}(c_1, c_2, \text{dept}) \]

  becomes

  \[ \text{Flight}(x_1, x_2, z_1, x_3, z_2) \leftarrow \text{Route}(x_1, x_2, x_3) \]

- Evaluate \( \varphi(x_1, \ldots, x_n, y_1, \ldots, y_m) \) in \( S \).

- For each tuple \( (a_1, \ldots, a_n, b_1, \ldots, b_m) \) that belongs to the result (i.e.

  \[ \varphi(a_1, \ldots, a_n, b_1, \ldots, b_m) \text{ holds in } S, \]

  do the following:
Canonical universal solution cont’d

• ... do the following:
  ◦ Create new (not previously used) null values \( \perp_1, \ldots, \perp_k \)
  ◦ Put tuples in target relations so that

\[
\psi(a_1, \ldots, a_n, \perp_1, \ldots, \perp_k)
\]

holds.

• What is \( \psi \)?

• It is normally assumed that \( \psi \) is a conjunction of atomic formulae, i.e.

\[
R_1(\bar{x}_1, \bar{z}_1) \land \ldots \land R_l(\bar{x}_l, \bar{z}_l)
\]

• Tuples are put in the target to satisfy these formulae
Canonical universal solution cont’d

- Example: no-direct-route airline:

\[ \text{Newroute}(x_1, z) \land \text{Newroute}(z, x_2) \leftarrow \text{Oldroute}(x_1, x_2) \]

- If \((a_1, a_2) \in \text{Oldroute}(a_1, a_2)\), then create a new null \(\bot\) and put:

\[ \text{Newroute}(a_1, \bot) \]
\[ \text{Newroute}(\bot, a_2) \]

into the target.

- Complexity of finding this solution: polynomial in the size of the source \(S'\):

\[ O(\sum \text{Evaluation of } \varphi \text{ on } S') \]
Canonical universal solution and conjunctive queries

- Canonical solution: $\text{CanSol}_M(S)$.
- We know that if $Q$ is a conjunctive query, then $\text{certain}_M(Q, S) = \text{certain}(Q, T)$ for every universal solution $T$ for $S$ under $M$.
- Hence
  \[
  \text{certain}_M(Q, S) = \text{certain}(Q, \text{CanSol}_M(S))
  \]

- Algorithm for answering $Q$:
  - Construct $\text{CanSol}_M(S)$
  - Apply naive evaluation to $Q$ over $\text{CanSol}_M(S)$
Beyond conjunctive queries

- Everything still works the same way for $\sigma$, $\pi$, $\Join$, $\cup$ queries of relational algebra. Adding union is harmless.
- Adding difference (i.e. going to the full relational algebra) is not.
- Reason: same as before, can encode validity problem in logic.
- Single rule, saying “copy the source into the target”

\[ T(x, y) :\leftarrow S(x, y) \]

- If the source is empty, what can a target be? Anything!
- The meaning of $T(x, y) :\leftarrow S(x, y)$ is

\[ \forall x \forall y \left( S(x, y) \rightarrow T(x, y) \right) \]
Beyond conjunctive queries cont’d

• Look at \( \varphi = \forall x \forall y (S(x, y) \rightarrow T(x, y)) \)

• \( S(x, y) \) is always false (\( S \) is empty), hence \( S(x, y) \rightarrow T(x, y) \) is true (\( p \rightarrow q \) is \( \neg p \lor q \))

• Hence \( \varphi \) is true.

• Even if \( T \) is empty, \( \varphi \) is true: universal quantification over the empty set evaluates to true:
  
  ○ Remember SQL’s \texttt{ALL}:

  ```
  SELECT * FROM R
  WHERE R.A > ALL (SELECT S.B FROM S)
  ```

  ○ The condition is true if \( \text{SELECT S.B FROM S} \) is empty.
Beyond conjunctive queries cont’d

• Thus if $S$ is empty and we have a rule $T(x, y) : \neg S(x, y)$, then all $T$’s are solutions.

• Let $Q$ be a Boolean (yes/no) query. Then

\[
certain_M(Q, S) = \text{true} \iff Q \text{ is valid}
\]

• Valid = always true.

• Validity problem in logic: given a logical statement, is it:
  - valid, or
  - valid over finite databases

• Both are undecidable.
Beyond conjunctive queries cont’d

• If we want to answer queries by rewritings, i.e. find a query $Q'$ so that
  \[ \text{certain}_M(Q, S) = Q'(\text{CANSO}_M(S)) \]
  then there is no algorithm that can construct $Q'$ from $Q$!
• Hence a different approach is needed.
Key problem

• Our main problem:
  Solutions are open to adding new facts

• How to close them?

• By applying the CWA (Closed World Assumption) instead of the OWA (Open World Assumption)
More flexible query answering: dealing with incomplete information

- Key issue in dealing with incomplete information:
  - Closed vs Open World Assumption (CWA vs OWA)
- CWA: database is closed to adding new facts except those consistent with one of the incomplete tuples in it.
- OWA opens databases to such facts.
- In data exchange:
  - we move data from source to target;
  - query answering should be based on that data and not on tuples that might be added later.
- Hence in data exchange CWA seems more reasonable.
Solutions under CWA – informally

- Each null introduced in the target must be justified:
  - there must be a constraint \( \ldots T(\ldots, z, \ldots) \ldots \leftarrow \varphi(\ldots) \) with \( \varphi \) satisfied in the source.

- The same justification shouldn't generate multiple nulls:
  - for \( T(\ldots, z, \ldots) \leftarrow \varphi(\bar{a}) \) only one new null \( \bot \) is generated in the target.

- No unjustified facts about targets should be invented:
  - assume we have \( T(x, z) \leftarrow \varphi(x) \), \( T(z', x) \leftarrow \psi(x) \) and \( \varphi(a), \psi(b) \) are true in the source.
  - Then we put \( T(a, \bot) \) and \( T(\bot', b) \) in the target but not \( T(a, \bot), T(\bot, b) \) which would invent a new “fact”: \( a \) and \( b \) are connected by a path of length 2.
How to formalize this – idea

Source-to-target dependencies of the form:

$$\psi_i(\bar{a}, z_1, \ldots, z_j, \ldots, z_k) \ :- \ \varphi_i(\bar{a}, \bar{b})$$

Justification for a null consists of:

- a dependency \((i)\)
- a witness \((\bar{a}, \bar{b})\) for \(\varphi_i(\bar{a}, \bar{b})\)
- a position \((j)\) of a null in the head of the rule.
Example

- Rule: \( \text{Flight}(c_1, c_2, z_1, \text{dept}, z_2) \rightarrow \text{Route}(c_1, c_2, \text{dept}) \)
- Witness: \( \text{Route}(\text{Edinburgh}, \text{Amsterdam}, 0600) \)
- This justifies up to two nulls:
  
  \[
  \text{Flight}(\text{Edinburgh}, \text{Amsterdam}, \bot_1, 0600, \bot_2)
  \text{ or }
  \text{Flight}(\text{Edinburgh}, \text{Amsterdam}, \bot, 0600, \bot)
  \]

- but not
  
  \[
  \text{Flight}(\text{Edinburgh}, \text{Amsterdam}, \bot_1, 0600, \bot_2)
  \text{ Flight}(\text{Edinburgh}, \text{Amsterdam}, \bot_3, 0600, \bot_4)
  \cdots
  \text{Flight}(\text{Edinburgh}, \text{Amsterdam}, \bot_{2n-1}, 0600, \bot_{2n})
  \]
Solutions under the CWA

• Each justification generates a null in $\text{CanSol}(S)$
• Hence for each solution $T$ under CWA there is a homomorphism

\[ h : \text{CanSol}(S) \rightarrow T \]

so that $T = h(\text{CanSol}(S))$
• The third requirement rules out tuples like

Flight(Edinburgh, Amsterdam, $\perp$, 0600, $\perp$)

• It invents a new fact: the same null is used twice in a tuple.
  ○ Not justified by the source and the rules
Solutions under the CWA

• The third requirement implies two facts:
  ◦ There is a homomorphism $h' : T \rightarrow \text{CanSol}(S)$
  ◦ $T$ contains the core of $T$

• What is the core?

• Suppose the Route relation has an extra attribute, in addition to source, destination, and departure time: it is flight#

• The same actual flight can have many flight numbers due to “code-sharing” so we might have
  
  \begin{align*}
  \text{Route(Edinburgh, Amsterdam, 0600, KLM 123)} \\
  \text{Route(Edinburgh, Amsterdam, 0600, AF 456)} \\
  \text{Route(Edinburgh, Amsterdam, 0600, CSA 789)}
  \end{align*}
Solutions under the CWA and cores cont’d

• The canonical solution then is:
  
  Flight(Edinburgh, Amsterdam, ⊥₁, 0600, ⊥₂)  
  Flight(Edinburgh, Amsterdam, ⊥₃, 0600, ⊥₄)  
  Flight(Edinburgh, Amsterdam, ⊥₅, 0600, ⊥₆)

• The core collapses it by means of a homomorphism
  
  \[ h(⊥₁) = h(⊥₃) = h(⊥₅) = ⊥₁ \quad h(⊥₂) = h(⊥₄) = h(⊥₆) = ⊥₂ \]

  to

  Flight(Edinburgh, Amsterdam, ⊥₁, 0600, ⊥₂)

• Core: A minimal subinstance \( T \) of \( \text{CANSol}(S) \) so that there is a homomorphism \( h : \text{CANSol}(S) \to T \)
Cores and CWA

- Cores are universal solutions too.
  - Advantage: space savings
  - Disadvantage: harder to compute
    - but still in polynomial time
- Basic fact: solutions under the CWA contain the core.
- Hence tuples such as
  
  \[
  \text{Flight(Edinburgh, Amsterdam, } \perp, 0600, \perp) \]

are disallowed.
Solutions under the CWA: summary

• There are homomorphisms

\[ h : \text{CanSol}(S) \rightarrow T \quad h' : T \rightarrow \text{CanSol}(S) \]

○ so that \( T = h(\text{CanSol}(S)) \)

• \( T \) contains the core of \( \text{CanSol}(S) \)
Query answering under the CWA

- Given
  - a source $S$,
  - a set of rules $M$,
  - a target query $Q$,

  A tuple $t$ is in $\text{certain}^{CWA}_M(Q, S)$ if it is in $Q(R)$ for every
  - solution $T$ under the CWA, and
  - $R \in \text{POSS}(T)$

- (i.e. no matter which solution we choose and how we interpret the nulls)
Query answering under the CWA – characterization

• Given a source $S$, a set of rules $M$, and a target query $Q$:
  \[ \text{certain}_{M}^{\text{CWA}}(Q, S) = \text{certain}(Q, \text{CanSol}(S)) \]

• That is, to compute the answer to query one needs to:
  ○ Compute the canonical solution $\text{CanSol}(S)$ – which has nulls in it
  ○ Find certain answers to $Q$ over $\text{CanSol}(S)$

• If $Q$ is a conjunctive query, this is exactly what we had before

• Under the CWA, the same evaluation strategy applies to all queries!
Query answering under the CWA cont’d

• Finding certain answers is possible for many classes of queries, e.g. for all relational algebra queries.

• \[
\text{Complexity of finding certain } M_{\text{CWA}}(Q, S) = \\
\text{complexity of finding certain answers to a query over a table with nulls}
\]

• polynomial time for conjunctive queries

• coNP-complete for relational algebra queries
CWA vs OWA: a comparison

• Recall the problematic case we had before:

\[ T(x, y) :– S(x, y) \]

• Possible targets are extensions of the source

• Hence finding certain answers to an arbitrary relational algebra query \( Q \) was undecidable.

• Under the CWA:
  ○ The only solution is a copy of \( S \) itself (and hence it is the canonical solution)
  ○ So certain answers to \( Q \) are just \( Q(S) \) – i.e. we copy \( S \), and evaluate queries over it, as suggested by the rule.
Data exchange and integrity constraints

• Integrity constraints are often specified over target schemas
• In SQL’s data definition language one uses keys and foreign keys most often, but other constraints can be specified too.
• Adding integrity constraints in data exchange is often problematic, as some natural solutions – e.g., the canonical solution – may fail them.

Plan:
  ○ review most commonly used database constraints
  ○ see how they may create problems in data exchange
Functional dependencies and keys

- **Functional dependency:**
  \[ X \rightarrow Y \]

  where \( X, Y \) are sequences of attributes. It holds in a relation \( R \) if for every two tuples \( t_1, t_2 \) in \( R \):
  \[
  \pi_X(t_1) = \pi_X(t_2) \quad \text{implies} \quad \pi_Y(t_1) = \pi_Y(t_2)
  \]

- **The most important special case:** keys

- **\( K \rightarrow U \),** where \( U \) is the set of all attributes:
  \[
  \pi_K(t_1) = \pi_K(t_2) \quad \text{implies} \quad t_1 = t_2
  \]

- That is, a key is a set of attributes that uniquely identify a tuple in a relation.
Inclusion constraints

• **Referential** integrity constraints: they talk about attributes of one relation but refer to values in another.

• An inclusion dependency

\[ R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n] \]

It holds when

\[ \pi_{A_1, \ldots, A_n}(R) \subseteq \pi_{B_1, \ldots, B_n}(S) \]
Foreign keys

• Most often inclusion constraints occur as a part of a foreign key

• Foreign key is a conjunction of a key and an ID:

$$R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n]$$ and

$$\{B_1, \ldots, B_n\} \rightarrow \text{all attributes of } S$$

• Meaning: we find a key for relation $S$ in relation $R$.

• Example: Suppose we have relations:
  Employee(EmplId, Name, Dept, Salary)
  ReportsTo(Empl1, Empl2).

• We expect both Empl1 and Empl2 to be found in Employee; hence:
  ReportsTo[Empl1] $\subseteq$ Employee[EmplId]
  ReportsTo[Empl2] $\subseteq$ Employee[EmplId].

• If EmplId is a key for Employee, then these are foreign keys.
Target constraints cause problems

• The simplest example:
  ◦ Copy source to target
  ◦ Impose a constraint on target not satisfied in the source

• Data exchange setting:
  ◦ \( T(x, y) :\neg S(x, y) \) and
  ◦ Constraint: the first attribute is a key

• Instance \( S: \)
  
  \[
  \begin{array}{cc}
  1 & 2 \\
  1 & 3 \\
  \end{array}
  \]

• Every target \( T \) must include these tuples and hence violates the key.
Target constraints: more problems

- A common problem: an attempt to repair violations of constraints leads to an sequence of adding tuples.

- Example:
  - Source $\text{DeptEmpl}(\text{dept\_id}, \text{manager\_name}, \text{empl\_id})$
  - Target
    - $\text{Dept}(\text{dept\_id}, \text{manager\_id}, \text{manager\_name})$,
    - $\text{Empl}(\text{empl\_id}, \text{dept\_id})$
  - Rule $\text{Dept}(d, z, n), \text{Empl}(e, d) \leftarrow \text{DeptEmpl}(d, n, e)$
  - Target constraints:
    - $\text{Dept}[\text{manager\_id}] \subseteq \text{Empl}[\text{empl\_id}]$
    - $\text{Empl}[\text{dept\_id}] \subseteq \text{Dept}[\text{dept\_id}]$
Target constraints: more problems cont’d

• Start with (CS, John, 001) in DeptEmpl.
• Put \text{Dept}(CS, \bot_1, John) \text{ and } \text{Empl}(001, CS) \text{ in the target}
• Use the first constraint and add a tuple \text{Empl}(\bot_1, \bot_2) \text{ in the target}
• Use the second constraint and put \text{Dept}(\bot_2, \bot_3, \bot_3') \text{ into the target}
• Use the first constraint and add a tuple \text{Empl}(\bot_3, \bot_4) \text{ in the target}
• Use the second constraint and put \text{Dept}(\bot_4, \bot_5, \bot_5') \text{ into the target}
• this never stops....
Target constraints: avoiding this problem

• Change the target constraints slightly:
  ◦ Target constraints:
    - $\text{Dept}[\text{dept}_\text{id}, \text{manager}_\text{id}] \subseteq \text{Empl}[\text{empl}_\text{id}, \text{dept}_\text{id}]$
    - $\text{Empl}[\text{dept}_\text{id}] \subseteq \text{Dept}[\text{dept}_\text{id}]$

• Again start with $(\text{CS}, \text{John}, 001)$ in $\text{DeptEmpl}$.

• Put $\text{Dept}(\text{CS}, \bot_1, \text{John})$ and $\text{Empl}(001, \text{CS})$ in the target

• Use the first constraint and add a tuple $\text{Empl}(\bot_1, \text{CS})$

• Now constraints are satisfied – we have a target instance!

• What’s the difference? In our first example constraints are very cyclic causing an infinite loop. There is less cyclicity in the second example. Bottom line: avoid cyclic constraints.