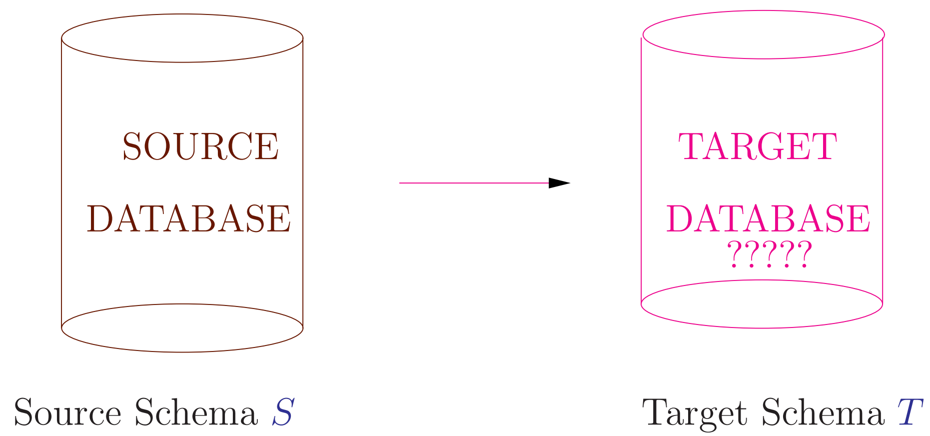
# Data exchange

- 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 transfered to data from organization $2$.
  - Queries need to be answered against the transferred data.

# Data Exchange


Source Schema $S$ Target Schema $T$

# Data Exchange



## Data exchange: an example

- We want to create a target database with the schema

  *Flight(city1,city2,aircraft,departure,arrival)*
  *Served(city,country,population,agency)*

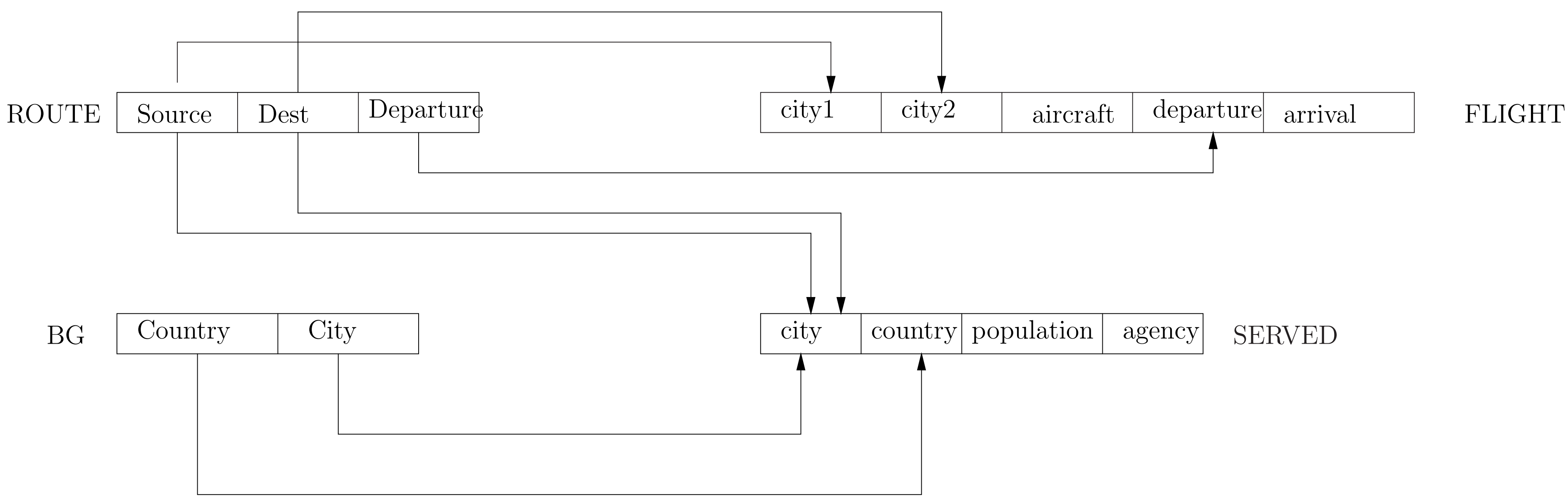
- We don't start from scratch: there is a source database containing relations

  *Route(source,destination,departure)*
  *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:

$$Flight(c1,\ c2,\ \_\_,\ dept,\ \_\_)\ :\!-\ Route(c1,\ c2,\ dept)$$

$$Served(city,\ country,\ \_\_,\ \_\_)\ :\!-\ Route(city,\ \_\_,\ \_\_),\ BG(country,\ city)$$

$$Served(city,\ country,\ \_\_,\ \_\_)\ :\!-\ Route(\_\_,\ city,\ \_\_),\ BG(country,\ city)$$

## Data exchange – targets

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

  *Flight(c1, c2, __, dept, __) :– Route(c1, c2, dept)*

  then it is satisfied by a source $S$ and a target $T$ if the constraint

$$\forall c_1, c_2, d \Big( \textit{Route}(c_1, c_2, d) \rightarrow \exists a_1, a_2 \ \big( \textit{Flight}(c_1, c_2, a_1, d, a_2) \big) \Big)$$

- This constraint is a relational calculus query that evaluates to *true* or *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:

ROUTE:

| Source | Destination | Departure |
|---|---|---|
| Edinburgh | Amsterdam | 0600 |
| Edinburgh | London | 0615 |
| Edinburgh | Frankfurt | 0700 |

BG:

| Country | City |
|---|---|
| UK | London |
| UK | Edinburgh |
| NL | Amsterdam |
| GER | Frankfurt |

Look at the rule

*Flight(c1, c2, __, dept, __)* :– *Route(c1, c2, dept)*

The right hand side is satisfied by

Route(Edinburgh, Amsterdam, 0600)

But what can we put in the target?

# Data exchange – targets

Rule: *Flight(c1, c2, _, dept, _) :– Route(c1, c2, dept)*

Satisfied by: Route(Edinburgh, Amsterdam, 0600)

Possible targets:

- Flight(Edinburgh, Amsterdam, $\perp_1$, 0600, $\perp_2$)
- Flight(Edinburgh, Amsterdam, B737, 0600, $\perp$)
- Flight(Edinburgh, Amsterdam, $\perp$, 0600, 0845)
- Flight(Edinburgh, Amsterdam, $\perp$, 0600, $\perp$)
- Flight(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, $\perp_1$, 0600, $\perp_2$) is most general as it makes no additional assumptions about the nulls.

# Towards good solutions

A solution is a database with nulls.
Reminder: every such database $T$ represents many possible complete databases, without null values:

Example

| A | B | C |
|---|---|---|
| 1 | 2 | $\perp_1$ |
| $\perp_2$ | $\perp_1$ | 3 |
| $\perp_3$ | 5 | 1 |
| 2 | $\perp_3$ | 3 |

$$v(\perp_1) = 4$$
$$v(\perp_2) = 3$$
$$v(\perp_3) = 5$$
$$\Longrightarrow$$

Semantics via valuations

| A | B | C |
|---|---|---|
| 1 | 2 | 4 |
| 3 | 4 | 3 |
| 5 | 5 | 1 |
| 2 | 5 | 3 |
| 3 | 7 | 8 |
| 4 | 2 | 1 |

$$\text{POSS}(T) = \{R \mid v(T) \subseteq R \text{ for some valuation } v\}$$

# Good solutions

- An optimistic view – A good solution represents ALL other solutions:

$$\mathsf{POSS}(T) \;=\; \{R \mid R \;\text{ is a solution without nulls}\}$$

- Shouldn't settle for less than – A good solution is at least as general as others:

$$\mathsf{POSS}(T) \supseteq \mathsf{POSS}(T') \;\text{ for every other solution } T'$$

- Good news: these two views are equivalent. Hence can take them as a definition of a good solutions.
- In data exchange, such solutions are called universal solutions.

# Universal solutions: another description

- A homomorphism is a mapping $h$ : Nulls $\longrightarrow$ Nulls $\cup$ 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

$$t = \text{Flight(Edinburgh, Amsterdam, } \perp_1, \text{0600, } \perp_2)$$

then

$$h(t) = \text{Flight(Edinburgh, Amsterdam, B737, 0600, 0845).}$$

- A solution is universal if and only if there is a homomorphism from it into every other solution.

# Universal solutions: still too many of them

- Take any $n > 0$ and consider the solution with $n$ tuples:

$$\begin{array}{lllll}
\text{Flight(Edinburgh,} & \text{Amsterdam,} & \perp_1, & 0600, & \perp_2) \\
\text{Flight(Edinburgh,} & \text{Amsterdam,} & \perp_3, & 0600, & \perp_4) \\
\ldots & & & & \\
\text{Flight(Edinburgh,} & \text{Amsterdam,} & \perp_{2n-1}, & 0600, & \perp_{2n})
\end{array}$$

- It is universal too: take a homomorphism

$$h'(\perp_i) \; = \; \begin{cases} \perp_1 & \text{if } i \text{ is odd} \\ \perp_2 & \text{if } i \text{ is even} \end{cases}$$

- It sends this solution into

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

# 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 $\perp_{i_1}, \ldots, \perp_{i_k}$ witness a conjunctive query, so do $h(\perp_{i_1}), \ldots, h(\perp_{i_k})$ — hence, one tuple suffices
- In general, if we have
  - a homomorphism $h : T \to T'$,
  - 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
  $$t \in Q(T) \quad \Leftrightarrow \quad t \in Q(T')$$
  because we have homomorphisms $T \to T'$ and $T' \to T$.

- Furthermore, if
  - $T$ is a universal solution, and $T''$ is an arbitrary solution,

  then
  $$t \in Q(T) \quad \Rightarrow \quad t \in Q(T'')$$

# Universal solutions and conjunctive queries cont'd

- Now recall what we learned about answering conjunctive queries over databases with nulls:
  - $T$ is a naive table
  - the set of tuples without nulls in $Q(T)$ is precisely certain$(Q,T)$ – certain answers over $T$
- Hence if $T$ is an arbitrary universal solution

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

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

$$\text{certain}_M(Q,S) \quad = \quad \text{certain}(Q,T)$$

  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:

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

  than

  $$\begin{array}{llll}
\text{Flight(Edinburgh, Amsterdam,} & \perp_1, & 0600, & \perp_2) \\
\text{Flight(Edinburgh, Amsterdam,} & \perp_3, & 0600, & \perp_4) \\
\ldots & & & \\
\text{Flight(Edinburgh, Amsterdam,} & \perp_{2n-1}, & 0600, & \perp_{2n})
\end{array}$$

- We now define a canonical universal solution.

# Canonical universal solution

- Convert each rule into a rule of the form:

  $$\psi(x_1, \ldots, x_n,\ z_1, \ldots, z_k) \quad :\!-\ \ \varphi(x_1, \ldots, x_n,\ y_1, \ldots, y_m)$$

  (for example,
  *Flight(c1, c2, __, dept, __) :– Route(c1, c2, dept)*

  becomes

  *Flight*($x_1,\ x_2,\ z_1,\ x_3,\ z_2$) :– *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.

$$R_1(\bar{x}_1, \bar{z}_1) \wedge \ldots \wedge R_l(\bar{x}_l, \bar{z}_l)$$

- Tuples are put in the target to satisfy these formulae

# Canonical universal solution cont'd

- Example: no-direct-route airline:

$$\text{Newroute}(x_1, z) \wedge \text{Newroute}(z, x_2) \ \text{:--}\ \text{Oldroute}(x_1, x_2)$$

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

  $\text{Newroute}(a_1, \perp)$
  $\text{Newroute}(\perp, a_2)$

  into the target.

- Complexity of finding this solution: polynomial in the size of the source $S$:

$$O(\sum_{\text{rules } \psi \text{ :- } \varphi} \text{Evaluation of } \varphi \text{ on } S)$$

# 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

$$\text{certain}_M(Q, S) \quad = \quad \text{certain}(Q, \text{CanSol}_M(S))$$

- 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 $\sigma, \pi, \bowtie, \cup$ 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"

$$T(x,y) \;:\!-\; S(x,y)$$

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

$$\forall x \forall y \; \big(S(x,y) \rightarrow T(x,y)\big)$$

# Beyond conjunctive queries cont'd

- Look at $\varphi = \forall x \forall y \, \big(S(x,y) \rightarrow T(x,y)\big)$
- $S(x,y)$ is always false ($S$ is empty), hence $S(x,y) \rightarrow T(x,y)$ is true ($p \rightarrow q$ is $\neg p \vee q$)
- Hence $\varphi$ is true.
- Even if $T$ is empty, $\varphi$ is true: universal quantification over the empty set evaluates to true:
  - Remember SQL's `ALL`:

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

  - The condition is true if `SELECT S.B FROM S` is empty.

# Beyond conjunctive queries cont'd

- Thus if $S$ is empty and we have a rule $T(x, y)$ :– $S(x, y)$, then all $T$'s are solutions.
- Let $Q$ be a Boolean (yes/no) query. Then

$$\text{certain}_M(Q, S) \;=\; \text{true} \quad\Leftrightarrow\quad Q \text{ is valid}$$

- 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

$$\text{certain}_M(Q, S) \quad = \quad Q'(\textsc{CanSol}_M(S))$$

  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 $\ldots T(\ldots, z, \ldots) \ldots$ :– $\varphi(\ldots)$ with $\varphi$ satisfied in the source.
- The same justification shouldn't generate multiple nulls:
  - for $T(\ldots, z, \ldots)$ :– $\varphi(\bar{a})$ only one new null $\bot$ is generated in the target.
- No unjustified facts about targets should be invented:
  - assume we have $T(x, z)$:– $\varphi(x)$, $\quad T(z', x)$:–$\psi(x)$ 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:

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

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: *Flight(c1, c2, z1, dept, z2) :– Route(c1, c2, dept)*
- Witness: Route(Edinburgh, Amsterdam, 0600)
- This justifies up to two nulls:

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

or

$$\text{Flight(Edinburgh, Amsterdam, } \perp, \text{ 0600, } \perp)$$

- but not

$$\begin{array}{l} \text{Flight(Edinburgh, Amsterdam, } \perp_1, \text{ 0600, } \perp_2) \\ \text{Flight(Edinburgh, Amsterdam, } \perp_3, \text{ 0600, } \perp_4) \\ \ldots \\ \text{Flight(Edinburgh, Amsterdam, } \perp_{2n-1}, \text{ 0600, } \perp_{2n}) \end{array}$$

# Solutions under the CWA

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

$$h : \textsc{CanSol}(S) \to T$$

  so that $T = h(\textsc{CanSol}(S))$

- The third requirement rules out tuples like

$$\text{Flight(Edinburgh, Amsterdam, } \perp\text{, 0600, } \perp\text{)}$$

- 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 $h' : T \to \textsc{CanSol}(S)$
  - $T$ 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
  Route(Edinburgh, Amsterdam, 0600, KLM 123)
  Route(Edinburgh, Amsterdam, 0600, AF 456)
  Route(Edinburgh, Amsterdam, 0600, CSA 789)

## Solutions under the CWA and cores cont'd

- The canonical solution then is:

  Flight(Edinburgh, Amsterdam, $\perp_1$, 0600, $\perp_2$)
  Flight(Edinburgh, Amsterdam, $\perp_3$, 0600, $\perp_4$)
  Flight(Edinburgh, Amsterdam, $\perp_5$, 0600, $\perp_6$)

- The core collapses it by means of a homomorphism

  $$h(\perp_1) = h(\perp_3) = h(\perp_5) = \perp_1 \quad h(\perp_2) = h(\perp_4) = h(\perp_6) = \perp_2$$

  to

  Flight(Edinburgh, Amsterdam, $\perp_1$, 0600, $\perp_2$)

- 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

$$\text{Flight}(\text{Edinburgh}, \text{Amsterdam}, \perp, 0600, \perp)$$

are disallowed.

# Solutions under the CWA: summary

- There are homomorphisms

$$h : \textsc{CanSol}(S) \to T \qquad\qquad h' : T \to \textsc{CanSol}(S)$$

  - so that $T = h(\textsc{CanSol}(S))$

- $T$ contains the core of $\textsc{CanSol}(S)$

# Query answering under the CWA

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

  a tuple $t$ is in

$$\mathsf{certain}_M^{\mathrm{CWA}}(Q, S)$$

  if it is in $Q(R)$ for every
  - solution $T$ under the CWA, and
  - $R \in \mathsf{POSS}(T)$
- (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 $Q$:

$$\text{certain}_M^{\text{CWA}}(Q, S) \quad = \quad \text{certain}(Q, \textsc{CanSol}(S))$$

- That is, to compute the answer to query one needs to:
  - Compute the canonical solution $\textsc{CanSol}(S)$ – which has nulls in it
  - Find certain answers to $Q$ over $\textsc{CanSol}(S)$
- 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.
- $$\text{Complexity of finding } \text{certain}_M^{\text{CWA}}(Q, S)$$
  $$=$$
  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:

$$T(x, y) \coloneq S(x, y)$$

- 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

$$\pi_{A_1,\ldots,A_n}(R) \quad \subseteq \quad \pi_{B_1,\ldots,B_n}(S)$$

# 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:

$$R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n] \quad \text{and}$$

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

- 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] ⊆ Employee[EmplId]
ReportsTo[Empl2] ⊆ 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:
  - $T(x, y) :- S(x, y)$ and
  - Constraint: the first attribute is a key
- Instance $S$:

| | |
|---|---|
| 1 | 2 |
| 1 | 3 |

- 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 DeptEmpl(dept_id,manager_name,empl_id)
  - Target
    - Dept(dept_id,manager_id,manager_name),
    - Empl(empl_id,dept_id)
  - Rule Dept$(d, z, n)$, Empl$(e, d)$ :– DeptEmpl$(d, n, e)$
  - Target constraints:
    - Dept[manager_id] $\subseteq$ Empl[empl_id]
    - Empl[dept_id] $\subseteq$ Dept[dept_id]

# Target constraints: more problems cont'd

- Start with (CS, John, 001) in DeptEmpl.
- Put Dept(CS, $\perp_1$, John) and Empl(001, CS) in the target
- Use the first constraint and add a tuple Empl($\perp_1$, $\perp_2$) in the target
- Use the second constraint and put Dept($\perp_2$, $\perp_3$, $\perp_3$') into the target
- Use the first constraint and add a tuple Empl($\perp_3$, $\perp_4$) in the target
- Use the second constraint and put Dept($\perp_4$, $\perp_5$, $\perp_5$') into the target
- this never stops....

# Target constraints: avoiding this problem

- Change the target constraints slightly:
  - Target constraints:
    - Dept[dept_id,manager_id] $\subseteq$ Empl[empl_id, dept_id]
    - Empl[dept_id] $\subseteq$ Dept[dept_id]
- Again start with (CS, John, 001) in DeptEmpl.
- Put Dept(CS, $\perp_1$, John) and Empl(001, CS) in the target
- Use the first constraint and add a tuple Empl($\perp_1$, 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.