Introduction to Databases

Exam of 05/07/2024

With Solutions

Diego Calvanese

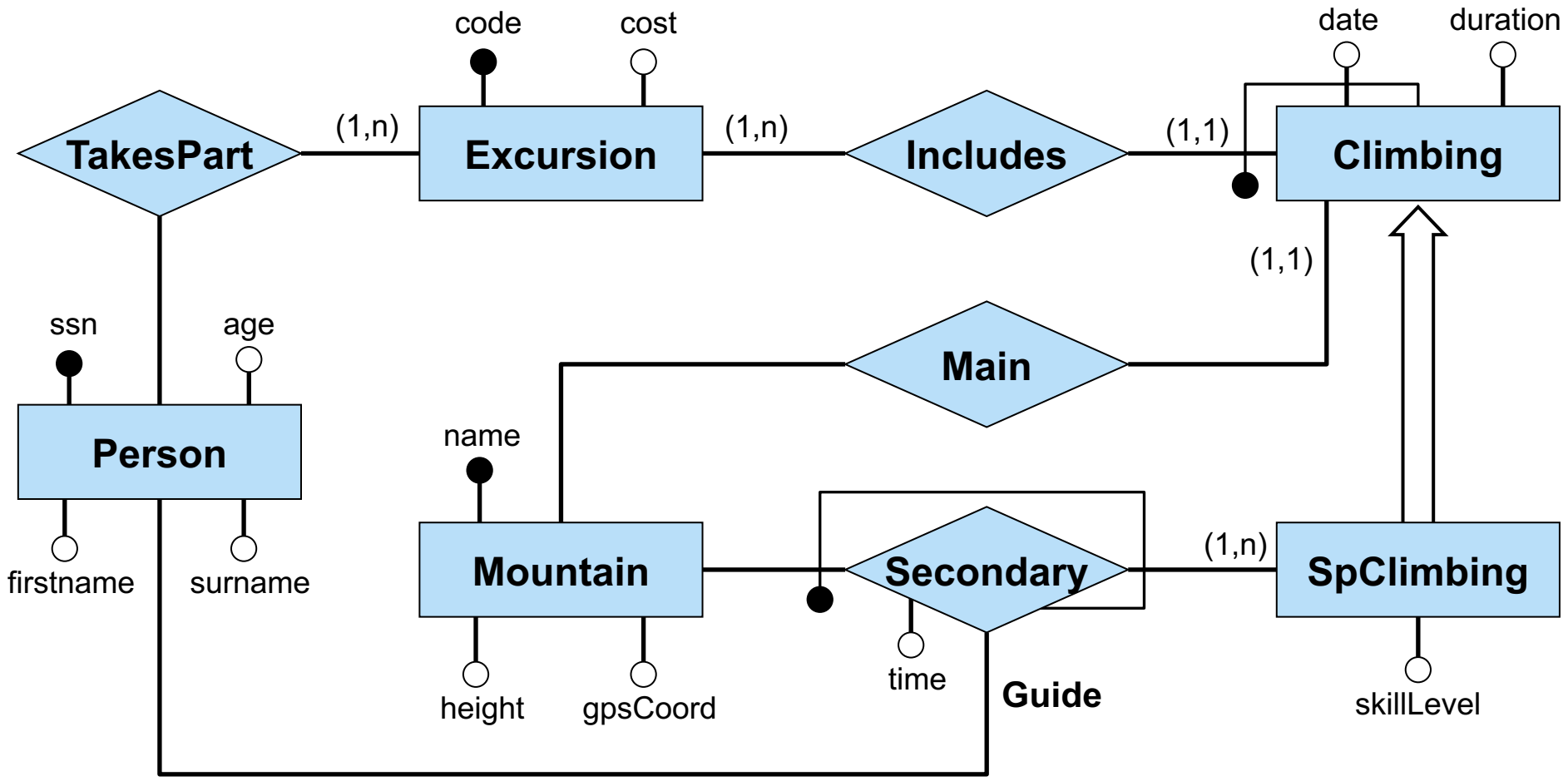
Bachelor in Computer Science
Faculty of Engineering
Free University of Bozen-Bolzano

http://www.inf.unibz.it/~calvanese/teaching/exams/idb/

Problem 1

Design the Entity-Relationship schema of an application for managing mountain excursions, for which the following information is of interest. Of each **excursion**, we are interested in the code (identifier), the cost, the persons who took part in the excursion (at least one), and the mountain climbings it includes (at least one). For each excursion, a maximum of one climbing per day takes place, and of each such **climbing** (which is specific to that excursion) we are interested in the date on which it took place, the duration, and the main mountain climbed. In addition, there are **special climbings**, for which we are interested also in the required skill level and the mountains (at least one) climbed in addition to the main one (these mountains are called secondary mountains of the climbing), each with the extra climbing time and the person who acted as guide in the climbing of that secondary mountain. Note that a person may not act as guide in more than one climbing of a secondary mountain on the same day (but they may on different days). Of each **mountain**, we are interested in the name (identifier), the height, and the GPS coordinates. Of each **person**, we are interested in the ssn (identifier), the first name, the surname, and the age.

Problem 1: Conceptual schema – Diagram



Note that the explicit identifier on the **Secondary** relationship is needed to take into account that the guide is determined by the special climbing and the mountain.

Problem 1: Conceptual schema – External constraints

For each instance I of the schema:

1. If (SpClimbing:c, Mountain:m, Guide:g) $\in$ *instances*(I,**Secondary**), then (Climbing:c, Mountain:m) $\notin$ *instances*(I,**Main**).

2. If (SpClimbing:c_1, Mountain:m_1, Guide:g) $\in$ *instances*(I,**Secondary**), (SpClimbing:c_2, Mountain:m_2, Guide:g) $\in$ *instances*(I,**Secondary**), (c_1,d_1) $\in$ *instances*(I,**date**), and (c_2,d_2) $\in$ *instances*(I,**date**), then $d_1 \neq d_2$.

Problem 2

Carry out the logical design of the database, producing the complete relational schema with constraints, taking into account the following indications.

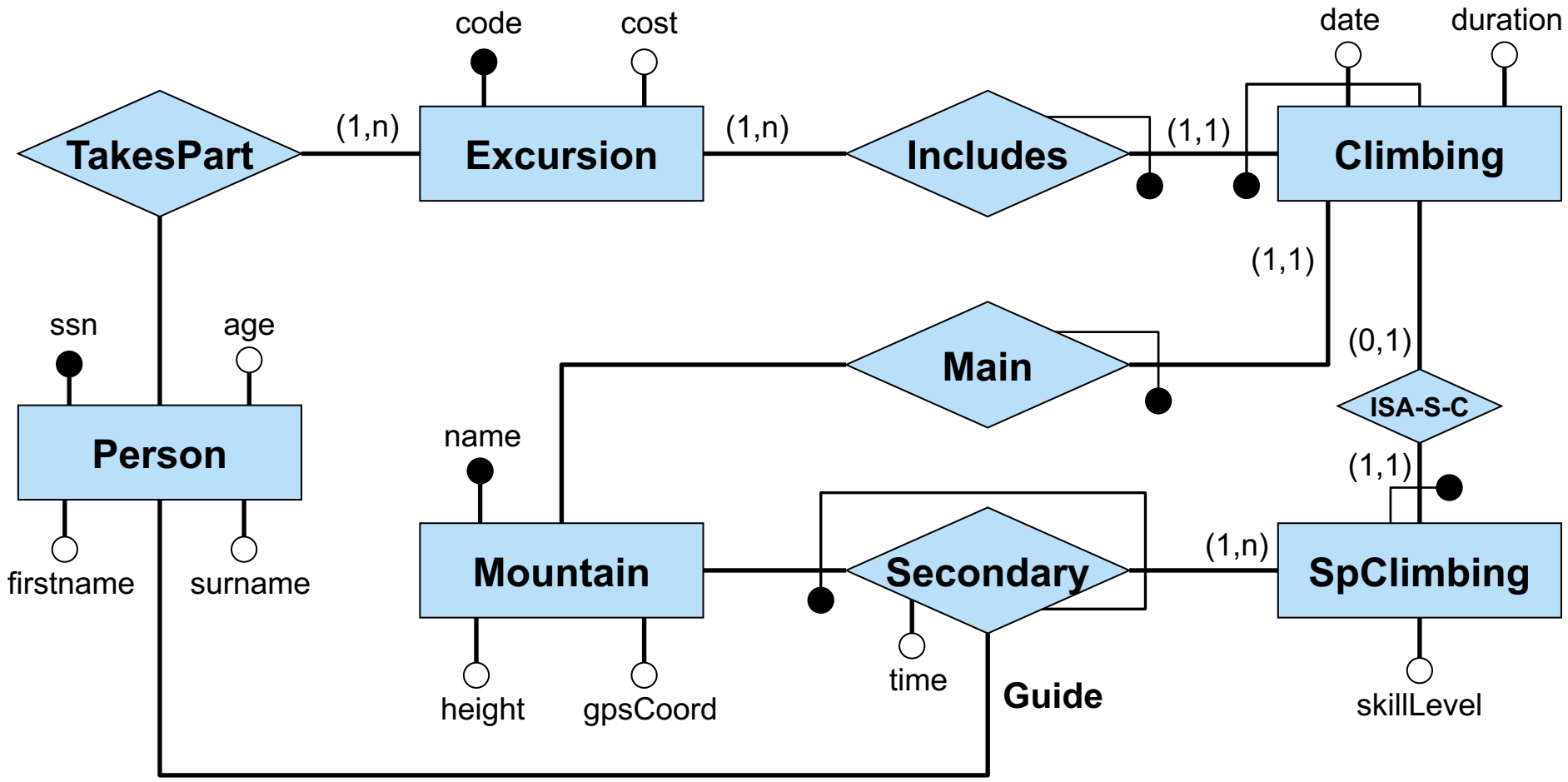
1. Every time we access a climbing, we are also interested in its main mountain.
2. Every time we access the information about a climbing, we always want to know whether it is a special climbing, and if so, we want to know the required skill level.

In your design you should follow the methodology adopted in the course, and you should produce:

- the restructured ER schema (possibly with external constraints),
- the direct translation into the relational model (possibly with external constraints), and
- the restructured relational schema (again with constraints).

You should motivate explicitly how the above indications affect your design.

Problem 2: Restructured conceptual schema



Problem 2: Restructured conceptual schema – External constraints

For each instance *I* of the schema:

1. If (SpClimbing:*s*, Mountain:*m*, Guide:*g*) ∈ *instances*(*I*,**Secondary**), and (SpClimbing:*s*, Climbing:*c*) ∈ *instances*(*I*,**ISA-S-C**), then (Climbing:*c*, Mountain:*m*) ∉ *instances*(*I*,**Main**).

2. If (SpClimbing:s_1, Mountain:m_1, Guide:*g*) ∈ *instances*(*I*,**Secondary**), (SpClimbing:s_2, Mountain:m_2, Guide:*g*) ∈ *instances*(*I*,**Secondary**), (SpClimbing:s_1, Climbing:c_1) ∈ *instances*(*I*,**ISA-S-C**), (SpClimbing:s_2, Climbing:c_2) ∈ *instances*(*I*,**ISA-S-C**), (c_1,d_1) ∈ *instances*(*I*,**date**), and (c_2,d_2) ∈ *instances*(*I*,**date**), then $d_1 \neq d_2$.

Problem 2: Direct translation (1/2)

Excursion(code, cost)
  inclusion: Excursion[code] $\subseteq$ Climbing[excursion]
  inclusion: Excursion[code] $\subseteq$ TakesPart[person]

Climbing(date, excursion, duration)
  foreign key: Climbing[excursion] $\subseteq$ Excursion[code]
  foreign key: Climbing[date,excursion] $\subseteq$ Main[climbing,excursion]

Person(ssn, firstname, lastname, age)

TakesPart(person, excursion)
  foreign key: TakesPart[person] $\subseteq$ Person[ssn]
  foreign key: TakesPart[excursion] $\subseteq$ Excursion[code]

SpClimbing(date, excursion, skillLevel)
  foreign key: SpClimbing[date,excursion] $\subseteq$ Climbing[date,excursion]
  inclusion: SpClimbing[date,excursion] $\subseteq$ Secondary[spClimbing,excursion]

Problem 2: Direct translation (2/2)

Mountain(<u>name</u>, height, gpsCoordinates)

Main(<u>date</u>, <u>excursion</u>, mountain)

foreign key: Main[date,excursion] $\subseteq$ Climbing[date,excursion]

foreign key: Main[mountain] $\subseteq$ Mountain[name]

Secondary(<u>date</u>, <u>excursion</u>, <u>mountain</u>, guide, time)

foreign key: Secondary[date,excursion] $\subseteq$ SpClimbing[date,excursion]

foreign key: Secondary[mountain] $\subseteq$ Mountain[name]

foreign key: Secondary[guide] $\subseteq$ Person[ssn]

key: date, guide

External constraint:

Main[date,excursion,mountain] $\cap$ Secondary[date,excursion,mountain] = $\emptyset$

Notice that the second external constraint has been translated into the additional key constraint on the relation **Secondary**.

Problem 2: Restructuring of the relational schema

1. Every time we access a climbing, we are also interested in its main mountain.
2. Every time we access the information about a climbing, we always want to know whether it is a special climbing, and if so, we want to know the required skill level.

- We take into account indication 1 by merging relation **Main** into **Climbing**.
- We take into account indication 2 by merging relation **SpClimbing** into **Climbing**. Notice that this requires making the attributes **skillLevel** nullable, and in fact we can use this attribute as a flag to indicate whether a climbing is special or not: a tuple in **Climbing** represents a special climbing if and only if the value of **skillLevel** is not NULL.

Problem 2: Restructured relational schema

We specify here only the relations with their constraints that have been changed with respect to the schema obtained through the direct translation.

The relations **Climbing**, **SpClimbing**, and **Main** are replaced by the following one:

Climbing(date, excursion, mountain, duration, skillLevel*)
  foreign key: Climbing[excursion] $\subseteq$ Excursion[code]
  foreign key: Climbing[mountain] $\subseteq$ Mountain[name]

The inclusion from **SpClimbing** to **Secondary** is replaced by the constraint on **Climbing**:

```
CHECK (skillLevel IS NULL OR
       (date,excursion) IN (SELECT date, excursion FROM Secondary))
```

The foreign key from **Secondary** to **SpClimbing** is replaced by the constraint on **Secondary**:

```
CHECK ((date,excursion) IN (SELECT date, excursion FROM Climbing
                            WHERE skillLevel IS NOT NULL))
```

The external constraint becomes:
  Climbing[date,excursion,mountain] $\cap$ Secondary[date,excursion,mountain] = $\emptyset$

Problem 3

Consider a database B containing the two tables: `Nodes(node)`, which stores all the nodes of a directed graph G, and `Edges(start,end)`, which stores all the edges of G, where an edge from node n_1 to node n_2 is represented by the tuple $t = \langle n_1, n_2 \rangle$ in the relation `Edges` for which t.`start` = n_1 and t.`end` = n_2.

If G contains the edge from n_1 to n_2, then n_2 is said to be a *successor* of n_1 in G and n_1 is said to be a *predecessor* of n_2 in G. Furthermore, a node is said to be a *source* if it has at least one successor and no predecessor.

We know that the database B satisfies both the foreign key constraint from `start` of `Edges` to `node` of `Nodes`, i.e., `Edges[start]` $\subseteq$ `Nodes[node]`, and the foreign key constraint from `end` of `Edges` to `node` of `Nodes`, i.e., `Edges[end]` $\subseteq$ `Nodes[node]`.

1. Write a **SQL** query that returns the average number of predecessors of the nodes of G (remember to consider also nodes having no predecessors).
2. Write a **relational algebra** query that computes all nodes of G that have as predecessors only source nodes (i.e., all nodes of G that have no predecessor that is not a source node).

Problem 3: Solution (1/2)

`Nodes(node)` `Edges(start,end)`

1. Write a **SQL** query that returns the average number of predecessors of the nodes of *G* (remember to consider also nodes having no predecessors).

```
WITH CountPred AS
   (SELECT N.node, COUNT(E.start) AS numPred
    FROM Nodes N LEFT JOIN Edges E ON N.node = E.end
    GROUP BY N.node)

SELECT AVG(numPred)
FROM  CountPred
```

Problem 3: Solution (2/2)

`Nodes(node)` `Edges(start,end)`

2. Write a **relational algebra** query that computes all nodes of *G* that have as predecessors only source nodes (i.e., all nodes of *G* that have no predecessor that is not a source node).

$$\text{Nodes} - \mathbf{REN}_{\text{node} \leftarrow \text{end}} (\mathbf{PROJ}_{\text{end}}(\text{Edges } \mathbf{JOIN}_{\text{start} = \text{node}} (\text{Nodes} - (\mathbf{REN}_{\text{node} \leftarrow \text{start}} (\mathbf{PROJ}_{\text{start}}(\text{Edges})) - \mathbf{REN}_{\text{node} \leftarrow \text{end}} (\mathbf{PROJ}_{\text{end}}(\text{Edges}))))))$$

Comments:

- $(\mathbf{REN}_{\text{node} \leftarrow \text{start}} (\mathbf{PROJ}_{\text{start}}(\text{Edges})) - \mathbf{REN}_{\text{node} \leftarrow \text{end}} (\mathbf{PROJ}_{\text{end}}(\text{Edges})))$ computes the source nodes.
- $(\text{Nodes} - \ldots)$ computes the nodes that are not source nodes.
- $\mathbf{REN}_{\text{node} \leftarrow \text{end}} (\mathbf{PROJ}_{\text{end}}(\text{Edges } \mathbf{JOIN}_{\text{start} = \text{node}} (\text{Nodes} - \ldots)))$ computes the nodes that have a predecessor that is not a source node.

Problem 4

Still referring to the database B described in Problem 3, suppose the foreign key constraint from `start` of `Edges` to `node` of `Nodes` is defined as `ON DELETE RESTRICT`, while the foreign key constraint from `end` of `Edges` to `node` of `Nodes` is defined as `ON DELETE CASCADE`.

Answer the following questions.

1. Is it always possible to obtain the deletion of any given tuple from the `Nodes` relation of graph *G* by executing only operations of the form ``DELETE FROM Nodes WHERE...`"? If the answer is positive, give reasons; if it is negative, say which nodes can be deleted and which cannot, again giving reasons.
2. Answer the question in the previous point when the graph *G* is a tree, which, remember, is a particular graph in which an edge from n_1 to n_2 indicates the hierarchical relationship between the parent node n_1 and the child node n_2.

Problem 4: Solution (1/2)

It is not always possible to obtain the deletion of any given tuple from the `Nodes` relation of graph *G* by executing only operations of the form ``DELETE FROM Nodes WHERE...''.

Specifically, the "`ON DELETE RESTRICT`" policy on the foreign key constraint from `start` of `Edges` to `node` of `Nodes` forbids the deletion from the `Nodes` relation of nodes that appear in the `start` column of the `Edges` relation, i.e., of nodes that have a successor. To delete such a node *n*, we first have to delete all its descendent nodes, i.e., all nodes reachable from *n* in *G*, starting from nodes that have no successor.

However, this is not possible if:

- the node *n* is part of a cycle in *G*, i.e., of a sequence n_1, ..., n_k of nodes such that *G* contains an edge from n_i to n_{i+1}, for all$1 \leq i \leq k$-1, and an edge from n_k to n_1, or
- from the node *n* one can reach in *G* a node that is part of a cycle in *G*.

In all other cases, we can obtain the deletion of a node *n* of *G* from the `Nodes` relation, by proceeding as explained in the following answer to question 2.

Problem 4: Solution (2/2)

When the graph *G* is a tree, ***it is always possible*** to obtain the deletion of any given tuple from the `Nodes` relation of *G* by executing only operations of the form ``DELETE FROM Nodes WHERE...''.

This is because a tree does not contain cycles. Therefore, it is always possible to obtain the deletion of a node *n* of *G*, by first deleting all its descendant nodes in the tree starting from the leaves, i.e., from those nodes reachable from *n* in *G* that have no successor. Indeed, the "`ON DELETE CASCADE`" policy on the foreign key constraint from `end` of `Edges` to `node` of `Nodes` ensures that, when a node *m* without successors is deleted from `Nodes`, also the edges to *m* are deleted from `Edges`. Hence, after deleting all leaf nodes, the nodes that had edges only to leaf nodes will become themselves leaf nodes, and therefore can be deleted. We can proceed deleting in this way, until the node *n* is deleted from `Nodes`.