Databases and Information Systems

ER Schema Restructuring

Prof. Ing. Lelio Campanile, PhD

Data Analytics Bachelor
Università degli Studi della Campania Luigi Vanvitelli

Logical Design — Part 1 of 2

Learning goals

- Understand why a conceptual ER schema must be **restructured** before being translated into the relational model
- Analyse and resolve **redundancies** using tables of volumes and accesses
- Remove **generalizations** by choosing the most appropriate strategy
- Apply **partitioning and merging** to improve operational efficiency
- Select **primary identifiers** for each entity

From Conceptual to Logical Design

The logical design process

Input

- The **conceptual ER schema** produced during conceptual design
- **Table of volumes**: how many instances of each entity and relationship
- **Table of operations**: which queries and updates, how often

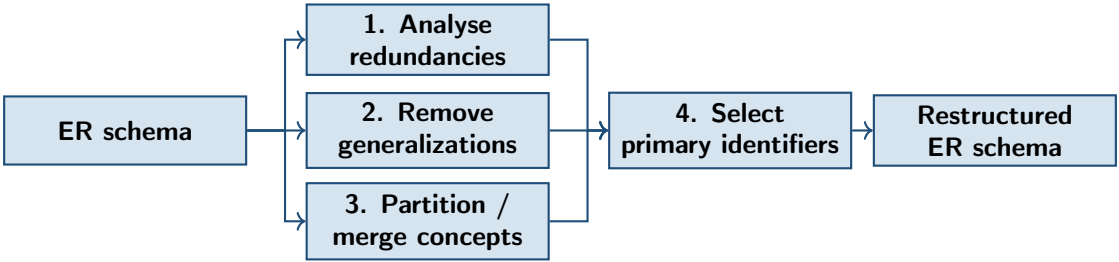
Two steps

1. **Restructuring** *(this lecture)*
   Optimise the schema before translation
2. **Translation**
   Map the restructured schema to relations

Why restructure?

Not all ER constructs translate naturally into relations, and some schema choices produce inefficient databases.

The four restructuring tasks



Each task is guided by the same two criteria: **cost of operations** (number of accesses) and **storage requirement** (number of bytes).

Step 1 — Analysis of Redundancies

What is a redundancy?

Definition

A **redundancy** is a piece of information that can be *derived* from other data already present in the schema — either by navigating relationships or by computing aggregates.

Examples of redundant attributes:

- `NumberOfInhabitants` on TOWN, when residents are stored in PERSON via a RESIDENCE relationship
- `TotalAmount` on PURCHASE, when line items are stored in COMPOSITION
- `TotalBalance` on CLIENT, derivable from linked ACCOUNTs

The trade-off

- **Advantage:** fewer accesses to read the derived value
- **Disadvantage:** extra writes to keep it up to date; extra storage

The decision must be **quantified** using volume and operation data.

Tools for the analysis: table of volumes

Definition

The **table of volumes** records the estimated number of **instances** (occurrences) of each entity and relationship in the database at a typical point in time.

Concept	Type	Volume
Town	E	200
Person	E	1 000 000
Residence	R	1 000 000

- **E** = Entity, **R** = Relationship
- Volumes are *estimates*, not exact counts
- They reflect the **steady-state** size of the database, not at import time
- Relationship volumes derive from entity volumes and cardinalities

Tools for the analysis: table of operations

Definition

The **table of operations** records each significant database operation, its type (**I**nteractive or **B**atch), and its expected **frequency**.

Operation	Type	Freq.
Op. 1: add person + town	I	500/day
Op. 2: print town data	I	2/day

- **I** (Interactive): latency-sensitive
- **B** (Batch): throughput-sensitive
- High-frequency operations dominate the cost
- Writes counted **twice** (read + write)

Tools for the analysis: table of accesses

Definition

The **table of accesses** counts, for each operation, how many entity and relationship occurrences must be visited — and whether each access is a **R**ead or a **W**rite.

Operation 1 (add person with town)

Concept	Type	Acc.	R/W
Person	E	1	W
Residence	R	1	W
Town	E	1	R

Operation 2 (print town data)

Concept	Type	Acc.	R/W
Town	E	1	R

Without redundancy, Op. 2 would also need to count all linked Residence/Person rows — up to 5 000 accesses.

Redundancy analysis: the PERSON–TOWN example

NoInhab is redundant: it can be derived by counting PERSON instances linked to TOWN via RESIDENCE.

Cost comparison (writes $\times 2$)

	With redund.	**Without**
Op. 1 (500/day)	$500 \times 4 = 2000$	$500 \times 2 = 1000$
Op. 2 (2/day)	$2 \times 1 = 2$	$2 \times 5001 \approx 10000$
Total	**2002**	**11000**

Decision

Maintaining the redundancy is **far cheaper**.
Keep `NoInhab`.

When to keep, when to remove

Keep the redundancy when...

- The derived data is **read very frequently**
- The source data changes **infrequently** (few updates)
- Recomputing it would require visiting **many occurrences**

Remove the redundancy when...

- The source data changes **very often** (high write cost)
- The derived value is **rarely read**
- Recomputing it is **cheap** (few accesses)

Always quantify the decision — intuition alone is unreliable.

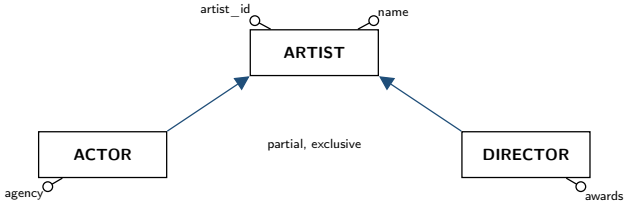
Convention: write accesses are counted **twice** in the total cost, because they require both reading the old value and writing the new one.

Step 2 — Removing Generalizations

Why generalizations must be removed

The problem

The **relational model has no native construct** for generalization hierarchies (also called ISA hierarchies in some traditions). Before translating, every generalization must be replaced with entities and relationships.

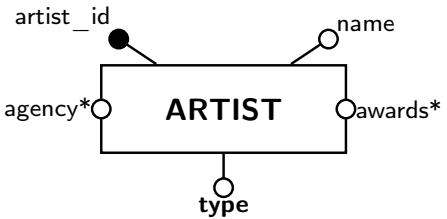


Three strategies are available. The right choice depends on the **type of generalization** (total/partial, exclusive/overlapping) and the **operations** that use the entities.

Option 1 — Collapse into the parent

How it works

Delete the child entities. Add their attributes to the parent, plus a **type discriminator** attribute.



When to use Option 1

- Operations treat all artists **uniformly** (no distinction between actors and directors)
- The child entities have **few specific attributes**
- Works for both **total and partial** generalizations — with a partial generalization, the type discriminator may be `null` for instances that belong to no child

Downside: child-specific attributes (`agency`, `awards`) will contain **NULL** for instances of the other type.

Option 2 — Collapse into the children

How it works

Delete the parent entity. Copy its attributes into each child entity.

When to use Option 2

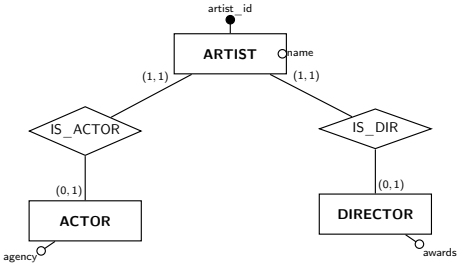
- The generalization is **total** (every artist is either an actor or a director — no instance belongs only to the parent)
- Operations **always distinguish** between actors and directors

Downside: parent attributes (`artist_id`, `name`) are **duplicated** in both tables. Queries that need all artists must combine the two relations (at the SQL level, via a `UNION` query).

Option 3 — Keep parent and children with relationships

How it works

Keep all three entities. Replace the generalization arrow with explicit **1:1 relationships** between parent and each child.



When to use Option 3

- The generalization is **partial** (some artists are neither actors nor directors) — the preferred case; cardinality on child side is $(0, 1)$
- Children have **many specific attributes** that should not pollute the parent
- Operations sometimes access **only the parent**, sometimes **only a child**

Note: Option 3 is also applicable with a **total** generalization; in that case the cardinality on the child side becomes $(1, 1)$ (every parent instance must participate in a child relationship).

Choosing the right option — summary

Condition	**Option 1** collapse into parent	**Option 2** collapse into children	**Option 3** keep all + rels
Generalization is **total**	possible	**preferred**	possible
Generalization is **partial**	possible	not valid	**preferred**
Children have **few attributes**	**preferred**	possible	possible
Children have **many attributes**	possible	possible	**preferred**
Ops always **distinguish** children	possible	**preferred**	preferred
Ops treat all instances **uniformly**	**preferred**	possible	possible

Options can be **combined**: apply Option 1 to one child and Option 3 to another if operations treat them differently.

Step 3 — Partitioning and Merging

The guiding principle

Core idea

Database accesses are minimised when the **data needed by an operation is stored together**, and when **data never accessed together is stored separately**.

Vertical partitioning

Split a single entity into two, separating attributes that are accessed by **different operations**.

Example: an EMPLOYEE entity whose administrative data (name, hire date) is read frequently, but whose financial data (salary, tax level) is accessed only by payroll runs.

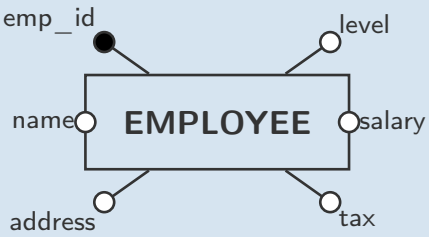
Merging

Combine two entities (or two relationships) into one when they are **always accessed together**.

Example: PASTTEACHING and CURRENTTEACHING describe the same semantic concept; operations never distinguish between them.

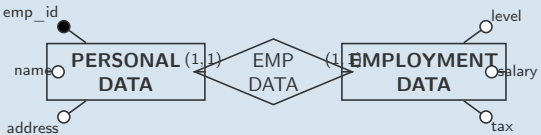
Vertical partitioning: example

Before partitioning



Personal data (name, address) is read by HR queries thousands of times a day. Financial data (salary, tax) is accessed only weekly by payroll.

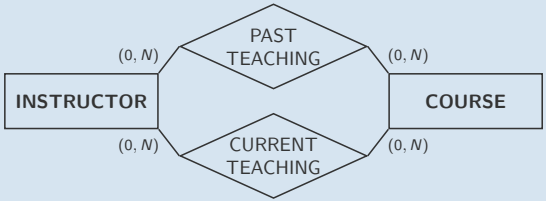
After partitioning



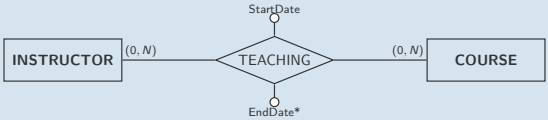
HR queries only touch PERSONAL DATA; payroll only touches EMPLOYMENT DATA. Fewer bytes read per access.

Merging relationships: example

Before merging



After merging



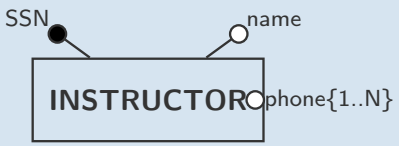
`EndDate` is null for current teaching assignments. One relationship instead of two; queries no longer need a UNION.

Removing multi-valued attributes

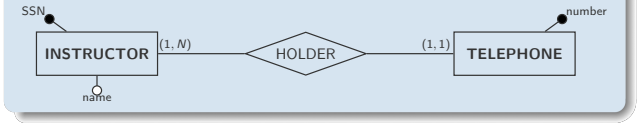
The problem

A **multi-valued attribute** (e.g. an instructor with multiple phone numbers) cannot be directly represented in the relational model. It must be converted during restructuring.

Before



After



The multi-valued attribute becomes a **new entity** linked by a 1:N relationship. Each phone number is now a separate tuple.

Step 4 — Selection of Primary Identifiers

Criteria for selecting primary identifiers

Goal

Every entity in the restructured schema must have exactly **one designated primary identifier**, which will become the primary key of the corresponding relation.

Prefer an identifier that:

- Never takes **NULL** values
- Uses **one or few attributes** (compact key)
- Is **internal** (not shared with another entity)
- Is used by **many operations** to access the entity

If no good candidate exists

Introduce a **surrogate key**: a new attribute (often called `code` or `id`) whose values are generated by the system solely to identify tuples.

Surrogate keys are compact, never NULL, and always internal.

Primary identifier selection: examples

TRAINEE entity

Two candidate identifiers:

- SSN (social security number): meaningful but long (many bytes) and may vary across countries
- Internal `code`: short, compact, never null

⇒ **Choose the internal code.**

SSN can be retained as an alternate key.

COURSE EDITION entity

External identifier: a course edition is identified by the combination of its `StartDate` and the parent COURSE entity.

- External identifiers make poor primary keys (composite, involve joins)
- Better: generate an `edition_code` from the course code

⇒ **Replace external id with a surrogate code.**

Summary

The four restructuring tasks — recap

1. **Analyse redundancies**
   Use volume and access tables to decide whether to keep or remove each derived attribute. Always quantify the trade-off.
2. **Remove generalizations**
   Choose among: collapse into parent (Option 1), collapse into children (Option 2), or keep all with relationships (Option 3). Base the choice on totality and operation patterns.
3. **Partition or merge concepts**
   Split attributes accessed by different operations; merge entities or relationships accessed together. Remove multi-valued attributes.
4. **Select primary identifiers**
   Prefer compact, non-null, internal identifiers. Introduce surrogate keys when needed.

How to reason during restructuring

1. List all **candidate redundancies** — attributes derivable from other data
2. For each, build the **access tables** with and without redundancy
3. Compare total costs (write accesses $\times 2$) and **decide**
4. For each **generalization**, check totality, overlapping, and operation patterns; pick Option 1, 2, or 3
5. Look for entity or relationship **pairs always accessed together** $\rightarrow$ merge
6. Look for entities whose attributes are **accessed by disjoint operations** $\rightarrow$ split
7. Replace all **multi-valued attributes** with new entities
8. For each entity, select or introduce the **primary identifier**

Questions for the class

- In the PERSON–TOWN example, what would change if Op. 2 (print town data) were executed 500 times per day instead of 2?
- Why is Option 2 (collapse into children) *not valid* when the generalization is partial?
- A student entity has two candidate identifiers: `matriculation_number` (unique within the university) and `fiscal_code` (unique nationwide). Which would you choose as primary identifier, and why?
- Can the same entity need both vertical partitioning *and* a surrogate key introduction? Give an example.
- What is the difference between merging two *entities* and merging two *relationships*?

In-class exercise

Consider the following scenario for a **streaming platform**:

The platform stores **users**, **content items**, and **genres**. Each user has a `username`, `email`, and `registration_date`. Content items have a `title`, `release_year`, and belong to exactly one genre. Each content item also stores a `view_count` (total number of times it has been played). Users can stream content; each streaming event is recorded with a `stream_date` and a `progress_pct`.

Tasks:

1. Identify a candidate **redundancy** in this schema and explain why it is redundant
2. The platform runs a "trending" query (top 10 most viewed content) **10 000 times per day**, but new stream events are recorded **5 000 times per day**. Using a simplified access analysis, decide whether to keep or remove the redundancy
3. The platform also stores content as either **MOVIE** or **SERIES**; series have an extra `episode_count`. Draw the generalization and choose the best removal option, justifying your answer