ER to Relational Mapping

Table of contents

Today's outline . . . 1
Why map at all? . . . 2
The (7+1) steps . . . 2
Step 1 — Entity mapping . . . 2
Step 2 — Weak entity mapping . . . 3
Question — Weak Entities (Cities/Provinces) . . . 3
Step 3 — Binary 1:1 relationship mapping . . . 4
Question — Binary Relationship . . . 4
Step 4 — Binary 1:N relationship mapping . . . 4
Question — Relationship Mapping . . . 5
Step 5 — Binary M:N relationship mapping . . . 5
Step 6 — Multivalued attribute mapping . . . 6
Step 7 — N-ary relationship mapping . . . 6
Step "+1" — Super/subclass mapping (EER) . . . 6
Final schema — Company database . . . 7
Worked example — Bank database . . . 7

Module 2, part 2 — continues on from 2026-03-09-the-relational-model-and-integrity-constraints[1]. See relational-mapping-notation[2] for the schema notation used throughout, and er-diagram-notation[3] for the ER/EER diagram notation being mapped *from*.

Today's outline

- The (7+1) steps for mapping an ER/EER diagram to a relational schema
- Worked example: the Company database
- Worked example: a Bank database

[1] courses/infs1200/2026-03-09-the-relational-model-and-integrity-constraints.pdf
[2] courses/infs1200/relational-mapping-notation.pdf
[3] courses/infs1200/er-diagram-notation.pdf

Why map at all?

The ER model is commonly used for **conceptual design** (user's perspective); the relational model is the basis for most commercial DBMSs (**storage perspective**). Mapping converts a conceptual ER schema into a logical relational schema — input: an ER model; output: relations with primary/foreign key constraints.

The (7+1) steps

1. Entity mapping
2. Weak entity mapping
3. Binary 1:1 relationship mapping
4. Binary 1:N relationship mapping
5. Binary M:N relationship mapping
6. Multivalued attribute mapping
7. N-ary relationship mapping
8. *(+1, for EER only)* Super/subclass mapping

Super/subclass mapping (step "+1") doesn't happen at one fixed point — it's generally done after step 1 or 2, whenever the subclass needs to exist for a later relationship-mapping step to reference it.

Running example — the **Company database**: `EMPLOYEE` (key `Ssn`, composite `Name` → `Fname`/`Mit`/`Lname`, `Sex`, `Salary`, `Address`, `DOB`), `DEPARTMENT` (keys `Dnumber`/`Dname`, derived `NumberOfEmployees`, multivalued `Locations`), `PROJECT` (keys `Pno`/`Pname`, `Plocation`), `DEPENDENT` (weak entity, partial key `DepName`, `Sex`, `DOB`, `Relationship`) — relationships `WORKSFOR` (`EMPLOYEE` N:1 `DEPARTMENT`), `MANAGES` (`EMPLOYEE` 1:1 `DEPARTMENT`, attribute `StartDate`), `CONTROLS` (`DEPARTMENT` 1:N `PROJECT`), `WORKSON` (`EMPLOYEE` M:N `PROJECT`, attribute `Hours`), `SUPERVISION` (`EMPLOYEE` recursive 1:N, roles *supervisor*/*supervisee*), `DEPENDENTSOF` (`EMPLOYEE` 1:N `DEPENDENT`, identifying), and `EMPLOYEE` disjointly (`d`) specialises into `SECRETARY` (`TypingSpeed`)/`ENGINEER` (`EngineerType`).

Step 1 — Entity mapping

For each (strong) entity type: create a relation, choose a key as the primary key, and include all simple attributes (not composite, derived, or multivalued).

```
Employee [ssn, fName, mIt, lName, dob, address, sex, salary]

Department [dNumber, dName]

Project [pNo, pName, pLocation]
```

`Name` isn't added directly (it's composite — its simple components `fName`/`mIt`/`lName` are added instead); `Locations` and `NumberOfEmployees` aren't added to `Department` yet (multivalued and derived respectively — derived attributes are *never* stored as columns, since they're computed on demand).

Step 2 — Weak entity mapping

For each weak entity type: create a relation; its primary key is the combination of its owner's primary key(s) and its own partial key; include a foreign key back to the owner's primary key; include all simple attributes.

```
Dependent [ssn, depName, sex, dob, relationship]
Dependent.ssn references Employee.ssn
```

`Ssn` (Employee's key) is included as part of `Dependent`'s own primary key, and `depName` is its partial key — so `Dependent`'s full key is `(ssn, depName)`.

If a weak entity has **more than one owner entity type**, its relation includes a foreign key to *each* owner's primary key, and its own primary key is the combination of all owner keys plus its partial key.

Question — Weak Entities (Cities/Provinces)

`PROVINCE` (key `name, premier`) —1:N `IN`→ `CITY` (partial key `name, mayor`). Resolving the dual use of "name" reasonably, which schema is the most reasonable translation?

A. `Cities [name, mayor]`, `Provinces [name, premier]`
B. `Cities [cName, pName, mayor]`, `Provinces [pName, premier]`, `Cities.cName references Provinces.pName`, `Cities.pName references Provinces.pName`
C. `Cities [cName, pName, mayor]`, `Provinces [pName, premier]`, `Cities.pName references Provinces.pName`
D. `Cities [cName, pName, mayor]`, `In [cName, pName]`, `Provinces [name, premier]`, `Cities.pName references Provinces.name`

> **i** Solution
>
> **(C).** `CITY` is a weak entity, so its real key is `(pName, cName)` — (A) is wrong since it drops `pName` from `Cities`' key entirely. (B) is wrong because `cName` (the partial key) is *not* a foreign key — only `pName` (inherited from the owner) is. (D) is wrong because `IN` is the *identifying relationship* for a weak entity — it does **not** get mapped to its own separate relation (unlike a regular 1:N relationship, which doesn't need one either, but D invents an unnecessary one here on top of misnaming `Provinces`' key column).

Step 3 — Binary 1:1 relationship mapping

For each binary 1:1 relationship: choose one participating entity type (prefer the one with **total participation**); include a foreign key from it back to the other entity's primary key; include the relationship's own simple attributes on the *chosen* side.

`MANAGES` (`EMPLOYEE` 1:1 `DEPARTMENT`, `DEPARTMENT` has total participation — every department must have a manager) — extend `Department`, since it's the side with total participation:

```
Department [dNumber, dName, mgrSSN, mgrStartDate]
Department.mgrSSN references Employee.ssn
```

Question — Binary Relationship

`S` —1:1 `R`→ `T` (`T` has attribute `b1`). Which schema is a reasonable translation?

A. `S [a1, b1]`, `T [b1]`, `S.b1 references T.b1`
B. `S [a1]`, `T [b1]`
C. `ST [a1, b1]`
D. `S [a1]`, `T [b1, a1]`, `T.a1 references S.a1`

> **i** Solution
>
> **(A).** `S` should be the side extended with the foreign key, since `S` has full/total participation in `R` (shown by the diagram's `1` cardinality directly against `S`, no partial-participation gap) — preferred over `T` for that reason. (B) loses all information about the relationship itself. (C) merges two intentionally-separate entity types into one relation, discarding a conceptual modelling decision. (D) puts the foreign key on the wrong side.

Step 4 — Binary 1:N relationship mapping

For each (non-weak) binary 1:N relationship: identify the entity type on the **N side**; add a foreign key there, referencing the primary key of the entity type on the **1 side**; include the relationship's simple attributes on the N side.

`WORKSFOR` (`EMPLOYEE` N:1 `DEPARTMENT`), `CONTROLS` (`DEPARTMENT` 1:N `PROJECT`), `SUPERVISION` (`EMPLOYEE` recursive, N side = *supervisee*):

```
Employee [ssn, fName, mIt, lName, dob, address, sex, salary, dNumber, superSSN]
Employee.dNumber references Department.dNumber
Employee.superSSN references Employee.ssn
```

```
Project [pNo, pName, pLocation, dNumber]
Project.dNumber references Department.dNumber
```

Question — Relationship Mapping

`A` —N `R` 1→ `B` —N `S` 1→ `C` (with attributes `a/d` on `A`, `b/e` on `B`, `c/f` on `C`). Which of the following appears in your relational schema: (A) `A [a, b, d]`, `A.b references B.b`; (B) `B [b, c, e]`, `B.c references C.c`; (C) `S [b, c]`; (D) all of the above; (E) none of the above?

i Solution

(B). Each 1:N relationship puts the foreign key on the **N side**: for `R` that's `A` (referencing `B`), and for `S` that's `B` (referencing `C`) — so `B [b, c, e]`, `B.c references C.c` is exactly right. (A) is wrong: `b` *is* the right foreign key column for `A`, but it shouldn't be part of `A`'s primary key — 1:N foreign keys are plain (non-key) attributes on the N side, not treated as part of a composite key like a weak entity's would be (the answer as written implies `b` joining `A`'s key, which the mapping rule doesn't call for). (C) is wrong — `S` is a relationship, not an entity type, so 1:N mapping doesn't create a relation for it at all. The full correct schema: `A [a, b, d]`, `A.b references B.b`; `B [b, c, e]`, `B.c references C.c`; `C [c, f]`.

Step 5 — Binary M:N relationship mapping

For each binary M:N relationship: create a **new** relation; include foreign keys to both participating entity types' primary keys (their combination becomes the new relation's primary key); include the relationship's own simple attributes.

`WORKSON` (`EMPLOYEE` M:N `PROJECT`, attribute `Hours`):

```
WorksOn [ssn, pNo, hours]
WorksOn.ssn references Employee.ssn
WorksOn.pNo references Project.pNo
```

Note: 1:1 and 1:N relationships *can* also be mapped this same "new relation" way — useful when the relationship is **sparse**, since it avoids `NULL` foreign key values that a direct-embedding mapping would otherwise produce for non-participating rows.

Step 6 — Multivalued attribute mapping

For each multivalued attribute: create a new relation; include a foreign key to the owning entity's primary key; the new relation's primary key is the combination of that foreign key and the multivalued attribute itself (if the multivalued attribute is composite, include its simple components instead).

`DEPARTMENT.Locations`:

```
DeptLocs [dNumber, location]
DeptLocs.dNumber references Department.dNumber
```

Step 7 — N-ary relationship mapping

For each N-ary relationship (excluding all-1-side cases): create a new relation; include foreign keys to all participating entity types; the foreign keys coming from the **many**-side entity types form the primary key; include the relationship's own simple attributes.

(No N-ary relationships in the Company database — `WORKSON` is binary.)

Step "+1" — Super/subclass mapping (EER)

Works for any combination of total/partial and disjoint/overlapping subclasses. For each subclass: create a relation; its primary key *is* the superclass's primary key; include a foreign key back to the superclass's relation; include the subclass's own simple attributes.

`EMPLOYEE` disjointly specialises into `SECRETARY`/`ENGINEER`:

```
Secretary [ssn, typingSpeed]
Secretary.ssn references Employee.ssn

Engineer [ssn, engineerType]
Engineer.ssn references Employee.ssn
```

This step doesn't happen at one fixed position in the 7-step sequence — it's usually done right after step 1 or 2 (whichever applies to the superclass), specifically so the subclass relations already exist by the time any later relationship-mapping step needs to reference them.

Final schema — Company database

```
Employee [ssn, fName, mIt, lName, dob, address, sex, salary, dNumber, superSSN]
Employee.dNumber references Department.dNumber
Employee.superSSN references Employee.ssn

Department [dNumber, dName, mgrSSN, mgrStartDate]
Department.mgrSSN references Employee.ssn

Project [pNo, pName, pLocation, dNumber]
Project.dNumber references Department.dNumber

Dependent [ssn, depName, sex, dob, relationship]
Dependent.ssn references Employee.ssn

Secretary [ssn, typingSpeed]
Secretary.ssn references Employee.ssn

Engineer [ssn, engineerType]
Engineer.ssn references Employee.ssn

WorksOn [ssn, pNo, hours]
WorksOn.ssn references Employee.ssn
WorksOn.pNo references Project.pNo

DeptLocs [dNumber, location]
DeptLocs.dNumber references Department.dNumber
```

Worked example — Bank database

A bank (unique `code`*,* `name`*,* `hoAddr`*) has branches (*`branchNo` *unique per-bank,* `addr`*). A branch has accounts (*`acNo`*,* `type`*,* `balance`*, 1 account holders) and loans (*`loanNo`*,* `type`*,* `amount`*, 1 loan holders). Customers (*`ssn`*,* `name`*,* `address`*) can hold accounts and/or loans.*

Working through the same 7 steps:

After step 1 (entity mapping):

```
Bank [code, name, hoAddr]

Account [acNo, type, balance]

Loan [loanNo, type, amount]
```

```
Customer [ssn, name, address]
```

After step 2 (BRANCH is a weak entity, owned by BANK):

```
Branch [bankCode, branchNo, addr]
Branch.bankCode references Bank.code
```

After step 4 (BRANCH 1:N ACCOUNT/LOAN — foreign keys added to the N side; note the FK is *composite* here, since Branch's own key is composite):

```
Account [acNo, type, balance, bankCode, branchNo]
Account.{bankCode, branchNo} references Branch.{bankCode, branchNo}

Loan [loanNo, type, amount, bankCode, branchNo]
Loan.{bankCode, branchNo} references Branch.{bankCode, branchNo}
```

After step 5 (ACCOUNT/LOAN M:N CUSTOMER — new relations):

```
AccountHolder [acNo, ssn]
AccountHolder.acNo references Account.acNo
AccountHolder.ssn references Customer.ssn

LoanHolder [loanNo, ssn]
LoanHolder.loanNo references Loan.loanNo
LoanHolder.ssn references Customer.ssn
```

Final schema:

```
Bank [code, name, hoAddr]

Branch [bankCode, branchNo, addr]
Branch.bankCode references Bank.code

Account [acNo, type, balance, bankCode, branchNo]
Account.{bankCode, branchNo} references Branch.{bankCode, branchNo}

Loan [loanNo, type, amount, bankCode, branchNo]
Loan.{bankCode, branchNo} references Branch.{bankCode, branchNo}
```

```
Customer [ssn, name, address]

AccountHolder [acNo, ssn]
AccountHolder.acNo references Account.acNo
AccountHolder.ssn references Customer.ssn

LoanHolder [loanNo, ssn]
LoanHolder.loanNo references Loan.loanNo
LoanHolder.ssn references Customer.ssn
```