ER to Relational Mapping

Table of contents

Why map at all . . . 2
The steps at a glance . . . 2
Step 1 — Strong entity types . . . 3
Step 2 — Weak entity types . . . 3
Step 3 — Binary 1:1 relationships . . . 4
Step 4 — Binary 1:N relationships . . . 5
Step 5 — Binary M:N relationships . . . 6
Step 6 — Multivalued attributes . . . 6
Step 7 — N-ary relationships . . . 7
Step +1 — Subclasses (EER) . . . 9
Final schema — Company database . . . 9
Second worked example — Bank database . . . 10

The (7+1)-step algorithm that converts a conceptual ER/EER diagram into a logical relational schema — input: an ER model; output: relations with primary and foreign key constraints. Introduced in 2026-03-16-er-to-relational-mapping[1] and practised in week5-tutorial-applied-class-4-entity-relationship-mapping[2] and week4-tutorial-case-study-3-dirt-road-driving[3]. See relational-mapping-notation[4] for the notation and naming style used throughout, er-diagram-notation[5] for the diagram being mapped *from*, and er-mapping-pitfalls[6] for the mistakes to check for afterwards.

[1] courses/infs1200/2026-03-16-er-to-relational-mapping.pdf
[2] courses/infs1200/week5-tutorial-applied-class-4-entity-relationship-mapping.pdf
[3] courses/infs1200/week4-tutorial-case-study-3-dirt-road-driving.pdf
[4] courses/infs1200/relational-mapping-notation.pdf
[5] courses/infs1200/er-diagram-notation.pdf
[6] courses/infs1200/er-mapping-pitfalls.pdf

Why map at all

The ER model is used for **conceptual design** — the user's perspective, good for talking to stakeholders about entities and relationships. The relational model is what commercial DBMSs actually implement — the storage perspective. Mapping is the bridge: every construct that the ER model can draw has to be re-expressed using only relations, primary keys[7] and foreign-keys[8], because those are the only tools the relational model has (see relational-model[9]).

The steps at a glance

#	Construct	Produces
1	Strong entity type	A new relation
2	Weak entity type	A new relation, keyed on owner's key + partial key
3	Binary 1:1 relationship	A foreign key on one side
4	Binary 1:N relationship	A foreign key on the N side
5	Binary M:N relationship	A new relation
6	Multivalued attribute	A new relation
7	N-ary relationship	A new relation — except 1:1:1 and N:1:1, which take foreign keys instead
+1	Subclass (EER)	A new relation per subclass

Only steps 1, 2, 5, 6, 7 and +1 create relations; steps 3 and 4 only add foreign keys to relations that already exist, and step 7 does the same in its two excluded cases. Working in order matters, because every later step needs the primary keys that earlier steps established.

The running example below is 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`), and the weak entity `DEPENDENT` (partial key `DepName`), with 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), `DEPENDENTSOF` (identifying), and `EMPLOYEE` disjointly specialising into `SECRETARY`/`ENGINEER`.

[7] courses/infs1200/relational-keys.pdf
[8] courses/infs1200/foreign-keys.pdf
[9] courses/infs1200/relational-model.pdf

Step 1 — Strong entity types

For each strong (non-weak) entity type: create a relation, choose one of its keys as the primary key, and include **all simple attributes**.

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

Department [dNumber, dName]

Project [pNo, pName, pLocation]
```

Three attribute kinds are deliberately not copied across as columns:

- **Composite** attributes contribute their simple components instead — `Name` never appears, `fName`/`mIt`/`lName` do.
- **Derived** attributes are never stored at all — `NumberOfEmployees` is computed on demand, so storing it would only create an opportunity for it to go stale.
- **Multivalued** attributes can't be stored in an atomic column and are deferred to step 6 — `Locations` is missing from `Department` for now.

Where an entity has several keys (`DEPARTMENT` has both `Dnumber` and `Dname`), one becomes the primary key and the others are simply ordinary attributes in the relational schema — see relational-keys[10].

Step 2 — Weak entity types

For each weak entity type: create a relation whose primary key is **the owner's primary key plus the weak entity's partial key**; add a foreign key from the inherited part back to the owner; include all simple attributes.

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

`Dependent`'s full key is `(ssn, depName)`: `ssn` is `Employee`'s key inherited through the identifying relationship, `depName` is the partial key (see relationship-constraints[11]). Note what does *not* happen — the identifying relationship `DEPENDENTSOF` gets no relation of its own; it is entirely represented by the inherited key.

If a weak entity has **more than one owner**, the rule generalises rather than changes: include as foreign keys the primary key attributes of *every* owner entity type, and make the primary

[10] courses/infs1200/relational-keys.pdf
[11] courses/infs1200/relationship-constraints.pdf

key the combination of all of them plus the partial key. For a weak `W` (partial key `a1`) owned by both `E1` (key `b1`) and `E2` (key `c1`):

```
E1 [b1]

E2 [c1]

W [a1, b1, c1]
W.b1 references E1.b1
W.c1 references E2.c1
```

All three columns are underlined as one key; both inherited keys are foreign keys in their own right.

When the owner's own key is composite, the foreign key is composite too. In the Bank example, `BRANCH` is a weak entity owned by `BANK` (`branchNo` is only unique within a bank):

```
Bank [code, name, hoAddr]

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

Step 3 — Binary 1:1 relationships

For each binary 1:1 relationship: pick one participating entity type, add a foreign key there referencing the other's primary key, and put the relationship's own simple attributes on that same chosen side.

Preference is given to entity types with total participation (relationship-constraints[12]) — it is part of the rule, not a stylistic tip. In the generic case, with `T` the totally-participating side (double line) and `c1` the relationship's own attribute:

```
S [a1, a2]

T [b1, b2, a1, c1]
T.a1 references S.a1
```

[12] courses/infs1200/relationship-constraints.pdf

If every department must have a manager but not every employee manages, then putting the foreign key on `Department` gives a value in every row, whereas putting it on `Employee` would leave it null for almost everyone (sql-nulls[13]). `MANAGES` is drawn with the double line on the `DEPARTMENT` side, so `DEPARTMENT` is the chosen side:

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

`mgrStartDate` is `MANAGES`'s own attribute, and travels with the foreign key. The foreign key is a plain attribute here, not part of `Department`'s key.

Both landed columns are **renamed** on arrival — `Employee.ssn` becomes `mgrSSN`, `StartDate` becomes `mgrStartDate` — and the same happens in step 4 (`Dnumber` → `dNumber` in `Employee`, `Ssn` → `superSSN` for `SUPERVISION`). Renaming an inherited attribute to say what role it plays in its new relation is expected, and is the standard fix when the imported name would collide with one already there.

Step 4 — Binary 1:N relationships

For each binary 1:N relationship that isn't an identifying relationship: find the entity type on the **N side**, add a foreign key there referencing the primary key of the entity type on the **1 side**, and put the relationship's own simple attributes on the N side too.

The direction is forced, not chosen: one row on the N side relates to exactly one row on the 1 side, so a single foreign key column can hold it. The other direction would need many values in one cell.

```
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
```

`SUPERVISION` is recursive, and is handled exactly like any other 1:N relationship — the N-side role (supervisee) carries a foreign key back to the 1-side role (supervisor) in the same relation.

[13] courses/infs1200/sql-nulls.pdf

Step 5 — Binary M:N relationships

For each binary M:N relationship: create a **new** relation; give it foreign keys to both participants' primary keys; **the combination of those foreign keys is its primary key**; add the relationship's own simple attributes.

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

A new relation is unavoidable here: neither side can hold a single foreign key value, since both sides relate to many rows of the other.

One wrinkle the Company example doesn't show: the relationship may bring a **key attribute of its own**, and if it does, that attribute joins the primary key rather than sitting outside it. For an M:N `R` between `S` and `T` carrying an underlined `c1` and a plain `c2`:

```
S [a1, a2]

T [b1, b2]

R [a1, b1, c1, c2]
R.a1 references S.a1
R.b1 references T.b1
```

`R`'s key is `(a1, b1, c1)` — the pair of foreign keys *plus* `c1`. This is how a relationship that can repeat between the same two entities (the same employee working on the same project in several different years, say) gets modelled without a duplicate-key clash.

1:1 and 1:N relationships *may* also be mapped this way, and it is worth doing when the relationship is **sparse** — a separate relation stores only the rows that actually participate, instead of leaving a null foreign key in every row that doesn't. In the standard implementation the foreign key column is mostly nulls, while the M:N implementation holds one short relation of just the participating pairs, and no nulls at all.

Step 6 — Multivalued attributes

For each multivalued attribute: create a new relation containing a foreign key to the owning entity's primary key plus the attribute itself; **the primary key is the combination of the two**. If the multivalued attribute is composite, include its simple components instead of the attribute itself.

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

Keying on both columns is what allows one department to have many locations while forbidding the same location twice. If the owning relation's key is itself composite, the foreign key here is composite — in the cinema example the snack relation carries all four columns of `Viewing`'s key:

```
ViewingSnack [name, id, date, time, snack]
ViewingSnack.{name, id, date, time} references Viewing.{name, id, date, time}
```

Step 7 — N-ary relationships

For each N-ary relationship type — **excluding the 1:1:1 and N:1:1 cases**, which are handled separately below: create a new relation; give it foreign keys to the primary key of **every** participating entity type; the foreign keys from the participants with **many cardinality** form the primary key; the relationship's own attributes may also contribute to that key; add its remaining simple attributes.

One relation body covers every non-excluded case, and only the underline moves. For a ternary `R` over `E1`, `E2` and `E3`, with own attributes `d1` and `d2`:

```
E1 [a1]

E2 [b1]

E3 [c1]

R [a1, b1, c1, d1, d2]
R.a1 references E1.a1
R.b1 references E2.b1
R.c1 references E3.c1
```

Cardinalities	Primary key of `R`
`N:M:P` — every participant many	All three foreign keys, `(a1, b1, c1)`
`N:M:1`	The two many-side foreign keys `(a1, b1)`; `c1` is a plain attribute
`N:M:1` with `d1` an underlined attribute of `R`	`(a1, b1, d1)` — the relationship's own key attribute joins them

Every participant is always represented, whatever its cardinality: a `1`-side foreign key is present but *outside* the key. For a ternary `K` over `H` (role `N`), `A` (role `M`) and `P` (role `1`), with own attribute `l`, that gives

```
K [b, i, q, l]
K.b references A.b
K.i references H.i
K.q references P.q
```

with only `b` and `i` underlined. The Company database contains no N-ary relationship — `WORKSON` is binary.

The excluded cases — 1:1:1 and N:1:1

These get **no new relation at all**. With at most one participant on a many side, there is no combination of foreign keys that would make a key, and the relationship can be absorbed exactly as in steps 3 and 4: one participating relation takes a foreign key to *each* of the others, plus the relationship's own attributes.

- **N:1:1** — the entity type on the `N` side is the one that takes them, and the choice is forced, since only its rows relate to exactly one row of each other participant:

```
E1 [a1, b1, c1, d1]
E1.b1 references E2.b1
E1.c1 references E3.c1

E2 [b1]

E3 [c1]
```

- **1:1:1** — nothing forces the direction, so *choose* a participant the same way step 3 does, preferring total participation, and put both foreign keys and the relationship's attributes there. The schema looks identical to the N:1:1 one above; only the reason for picking `E1` differs.

The giveaway that a question is testing this exclusion is a ternary diamond with at most one `N` on its edges: count the many sides before reaching for a new relation, because with fewer than two of them there is nothing to key it on.

Step +1 — Subclasses (EER)

For each subclass (specialisation-and-generalisation[14]): create a relation whose primary key **is** the superclass's primary key; add a foreign key from it back to the superclass relation; include the subclass's own simple attributes. The same rule works for every combination of total/partial and disjoint/overlapping specialisation.

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

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

A subclass with no attributes of its own still gets a relation — it is the row's *membership* in that subclass that is being recorded.

This step has no fixed position in the sequence. It is done straight after step 1 or 2 (whichever created the superclass), specifically so the subclass relations exist by the time a later relationship step needs to reference one — and relationships drawn to a subclass must reference the *subclass* relation, not the superclass.

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]
```

[14] courses/infs1200/specialisation-and-generalisation.pdf

```
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
```

Eight relations: the three strong entity types became three, the weak entity one, the two subclasses two, the M:N relationship one, and the multivalued attribute one — while all four of the 1:1 and 1:N relationships (`MANAGES`, `WORKSFOR`, `CONTROLS`, `SUPERVISION`) became nothing more than foreign key columns on relations that already existed.

Second worked example — Bank database

The other complete run-through is the bank: a `BANK` (unique `Code`, `Name`, `HOAddr`) has several branches; a `BANK-BRANCH` has a `BranchNo` unique only within its bank, so it is **weak**. One branch has many `ACCOUNT`s (`ACNo`, `Type`, `Balance`) and many `LOAN`s (`LoanNo`, `Type`, `Amount`), and every account and loan has one or more holders drawn from `CUSTOMER` (`SSN`, `Name`, `Addr`) — two M:N relationships, `AHOLDER` and `LHOLDER`.

```
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
```

Two things are worth reading off this schema. First, a weak entity's composite key propagates: because `Branch` is keyed on `(bankCode, branchNo)`, every 1:N foreign key drawn *from* `Branch` is composite too. Second, it propagates as a plain column — `Account` has its own key `acNo`, so `bankCode` and `branchNo` sit outside `Account`'s primary key, exactly as step 4 requires. `AccountHolder` and `LoanHolder`, by contrast, have no key of their own and are keyed on their pair of foreign keys.

The result is a logical schema, ready to be implemented with sql-ddl[15] and checked for redundancy with normal-forms[16]. Whether each inherited foreign key is *part of* its relation's primary key — the question that decides most of the answers above — is tabulated in foreign-keys[17].

[15] courses/infs1200/sql-ddl.pdf
[16] courses/infs1200/normal-forms.pdf
[17] courses/infs1200/foreign-keys.pdf