Foreign Keys

Table of contents

Definition 1
Shapes a foreign key can take 3
Is the foreign key part of the primary key? 5
Nulls, and when to avoid them 6
Enforcement 6

A **foreign key** is a set of attributes in one relation that links it to another relation's primary key — the only mechanism the relational model has for expressing a relationship between two relations, and therefore the thing almost every mapping step in er-to-relational-mapping[1] produces. Introduced in 2026-03-09-the-relational-model-and-integrity-constraints[2]; see relational-keys[3] for keys *within* a relation and relational-mapping-notation[4] for the `references` notation used throughout.

Definition

Let `FK` be a set of attributes in relation `R1`, and `PK` the primary key of relation `R2`. `FK` is a foreign key referencing `PK` if:

1. `FK` and `PK` are drawn from the same domain(s) — the names need not match, only the domains (see relational-model[5] on why attribute names and domains are independent), and
2. for every tuple t_1 in `R1`, either $t_1[FK]$ is `NULL`, or there exists a tuple t_2 in `R2` with $t_1[FK] = t_2[PK]$.

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

Condition 2 is the **referential integrity constraint**. Note what it permits: a foreign key value may be `NULL` (the row simply doesn't participate), but it may never be a value that points nowhere. A dangling reference is the violation; an absent reference is not.

```
Department [id, name, manager]
Department.manager references Employee.id

Employee [id, name, sex, salary, department]
Employee.department references Department.id
```

Foreign keys can point in both directions between the same pair of relations, as here — `Department.manager` says who runs the department, `Employee.department` says who works in it. They are two separate constraints expressing two separate relationships.

An instance makes the permission concrete: `Department` 4 (Human Resources) has a `NULL` manager and employee 2034 a `NULL` department, and both rows are legal. An employee whose `department` read 5, with no department 5 in the referenced relation, would not be.

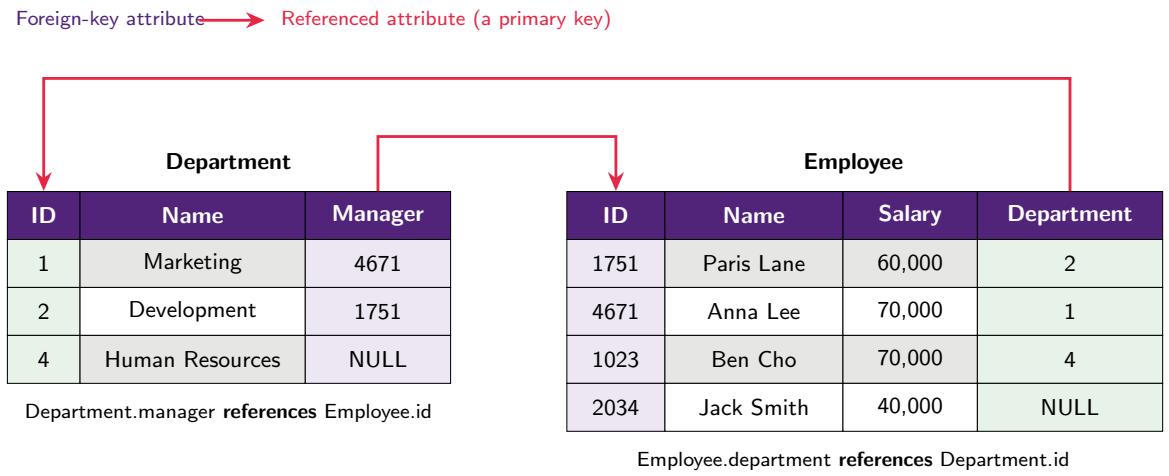


Figure 1: A two-way foreign key between Department and Employee. Each arrow starts at the foreign-key column and its head lands on the referenced primary key — the direction never varies in this course. The tinted columns pair each foreign key with the primary key it points at; the two NULLs are legal, because a foreign key may be absent but may never point nowhere.

Shapes a foreign key can take

Self-referencing

A relation can reference itself, because the two attributes involved share a domain even though they play different roles. Adding a `managerId` column to the same `Employee` relation gives one:

```
Employee [id, name, salary, department, managerId]
Employee.managerId references Employee.id
```

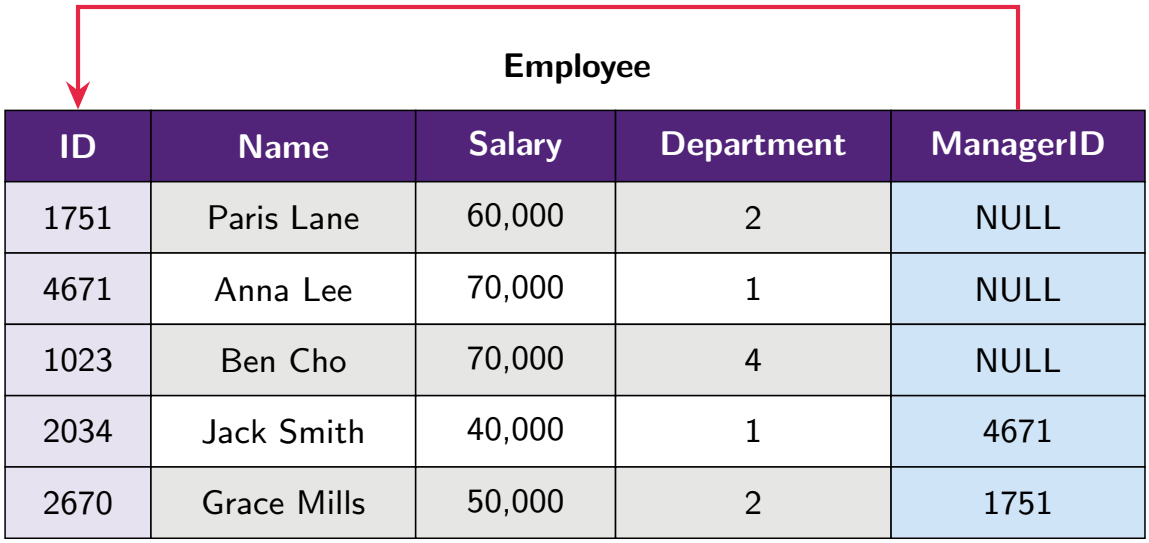


Figure 2: A self-referencing foreign key: the arrow leaves ManagerID and returns to ID in the same relation. The three top-level employees carry NULL, which is what stops the reference chain rather than a special case in the constraint.

The company schema does the same thing with different names:

```
Employee [ssn, fName, lName, dob, superSSN]
Employee.superSSN references Employee.ssn
```

This is what a recursive relationship (relationship-types[6] — `SUPERVISION`: an employee supervises many employees) maps to: the *N*-side role, the supervisee, carries the foreign key back to the *1*-side role, the supervisor. It is also what a sql-joins[7] self-join follows back.

Composite

If the referenced primary key is composite, the foreign key must be composite too, and the whole thing is **one** constraint over a tuple of columns:

```
Student [sid, name]

Course [cid, department, manager]

Enrollment [sid, cid, department, grade]
Enrollment.sid references Student.sid
Enrollment.{cid, department} references Course.{cid, department}
```

`Course`'s key is the pair `(cid, department)`, so `Enrollment` must carry both columns to name one course, and its own key is all three of `(sid, cid, department)`.

Splitting a composite foreign key into several single-column constraints is a real and commonly-marked mistake — see er-mapping-pitfalls[8] — because it no longer forces all the columns to jointly match *one* referenced row.

Two foreign keys to the same relation

When a relationship uses the same entity type in two different roles, the referencing relation gets two separate foreign keys to it, given distinct column names so the roles stay readable:

```
PeerEvaluation [studentSid, assessmentSid, number, evaluationId, sidGrades, mark]
PeerEvaluation.studentSid references Student.sid
PeerEvaluation.sidGrades references Student.sid
PeerEvaluation.{assessmentSid, number} references Assessment.{sid, number}
```

`studentSid` is the evaluator submitting the mark; `sidGrades` is the student whose assessment is being marked.

[6] courses/infs1200/relationship-types.pdf
[7] courses/infs1200/sql-joins.pdf
[8] courses/infs1200/er-mapping-pitfalls.pdf

Is the foreign key part of the primary key?

This is the question that decides most mapping answers, and the rule is structural rather than a matter of taste:

Situation	Foreign key is…
1:N relationship, FK on the N side (an entity with its own key)	A plain, non-key attribute
1:1 relationship, FK on the chosen side	A plain, non-key attribute
Weak entity, FK to its owner	*Part of* the weak entity's composite primary key
M:N relationship's own relation	Part of the primary key (the pair of FKs *is* the key)
N-ary relationship's relation, FK from a *many*-side participant	Part of the primary key
N-ary relationship's relation, FK from a *1*-side participant	A plain, non-key attribute
Multivalued attribute's relation	Part of the primary key (with the value itself)
Subclass relation	*Is* the primary key — the subclass's key is the superclass's key

So in `Employee [ssn, …, dNumber, superSSN]` the department and supervisor foreign keys are ordinary columns, whereas in `Dependent [ssn, depName, …]` the inherited `ssn` is underlined as part of the key. Treating an ordinary 1:N foreign key as key material is a standard wrong answer.

The dividing line is whether the referencing entity has a key of its own. An entity that doesn't — the cinema `Viewing`, identified only by *which* movie, *which* customer and *when* — is being identified through its owners, so the inherited foreign keys do join its key: `Viewing [name, id, date, time, attendant]`, keyed on all four of `(name, id, date, time)`. That reads as weak-entity behaviour with two owners rather than as an exception to the rule, and the *shape* is straight out of the mapping rules: a weak entity with more than one owner takes a foreign key to each owner and is keyed on every owner key plus its partial key (step 2 of er-to-relational-mapping[9]). Reading `Viewing` as a weak entity with two owners or as a strong entity whose key happens to be composite produces the same relation, so the mapping answer is the same either way.

[9] courses/infs1200/er-to-relational-mapping.pdf

Nulls, and when to avoid them

Because a foreign key may be `NULL`, embedding a 1:1 or 1:N relationship directly into an existing relation produces a null in every row that doesn't participate. When a relationship is **sparse** — few of the rows actually take part — mapping it as its own relation instead (the M:N treatment) avoids all those nulls at the cost of an extra join. That trade-off is noted under step 5 of er-to-relational-mapping[10]; see sql-nulls[11] for what nulls then do to queries.

Enforcement

Referential integrity is declarative — the DBMS rejects any insert, update, or delete that would leave a dangling reference, with no application code involved (see integrity-constraints[12]).

Three operations from week4-tutorial-applied-class-3-relational-integrity[13] show the full range, and they are worth reading together because two of them look dangerous and are not:

Operation on the company schema	Verdict
Delete the `EMPLOYEE` tuple with `ssn = 987654321`	**Violates referential integrity** — tuples in `WORKS_ON`, `DEPENDENT`, `DEPARTMENT` *and* `EMPLOYEE` itself all reference it
Modify `EMPLOYEE.superSSN` of `999887777` to `943775543`	**Violates referential integrity** — no employee has that `ssn`; an update to a foreign key is checked exactly like an insert
Delete the `WORKS_ON` tuples with `essn = 333445555`	**No violation** — nothing references `WORKS_ON`, so removing rows from it cannot dangle anything

The last row is the shape people get wrong: whether a delete is safe depends on what points *at* the relation, not on how many foreign keys the relation itself carries. `WORKS_ON` is all foreign key and is still free to delete from. The `EMPLOYEE` delete is dangerous precisely because four relations, one of them `EMPLOYEE` itself through `superSSN`, point back at it.

What happens to *dependent* rows when a referenced row is deleted or its key updated is a policy choice attached to the foreign key itself: the violation can be **rejected**, **cascaded**, or the referencing attribute values **modified to a default value**. See update-operations-and-constraint-violations[14] for those conceptual options and sql-ddl[15] for the four named `ON DELETE` / `ON UPDATE` actions that spell them out in SQL.

[10] courses/infs1200/er-to-relational-mapping.pdf
[11] courses/infs1200/sql-nulls.pdf
[12] courses/infs1200/integrity-constraints.pdf
[13] courses/infs1200/week4-tutorial-applied-class-3-relational-integrity.pdf
[14] courses/infs1200/update-operations-and-constraint-violations.pdf
[15] courses/infs1200/sql-ddl.pdf