"Relations for Relationships"

This note might help those who have struggled with M359's so-called "relation for relationship" method of representing, in a relational database design, something that is shown as a relationship in an E-R model.

Preamble

It is useful to reflect on why the term *relation* was introduced—by logicians, long before the idea of relational databases came about—for the formal construct that M359 is (almost) all about. And the reason is indeed the close connection in English between the words *relation* and *relationship*.

The following definitions are from the Shorter Oxford English Dictionary (SOED):

> **relation** *n.*
>
> 2 The existence or effect of a connection, correspondence, or contrast between things; the particular way in which one thing stands in connection with another; any connection or association conceivable as naturally existing between things.
>
> **relationship** *n.*
>
> The state or fact of being related; a connection, an association, **...**

If you read about relations in any introductory text on logic, you will probably find that the term is used specifically for what we now call binary relations—relations of degree two, i.e., having two attributes in our terminology. And familial relationships such as "*a* is a parent of *b*", "*a* is a sibling of *b*" and so on are often used as examples.

> **Reminder:** "*a* is a parent of *b*" is an example of a ***predicate*** in two variables (*a* and *b*). The corresponding relation is the set of all pairs of *a* and *b* values that ***satisfy*** the predicate. Such a pair satisfies the predicate if the result of substituting them for the variables in the predicate is a true statement (a true ***proposition***).

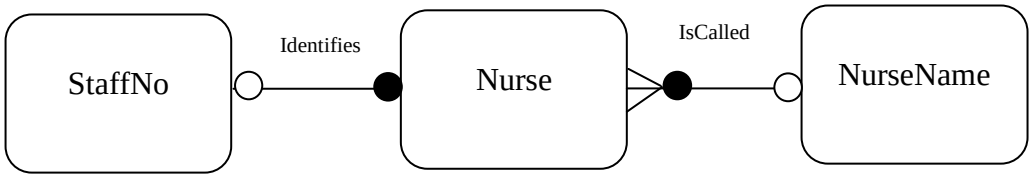
But some relations involve more than two things. For example, that I am your tutor on M359 involves two people (me and you) and a course: three things altogether. *Aside:* Moreover, the relation here ***irreducibly*** involves all three things: that *t* is *s*'s tutor on course *c* doesn't always follow from the facts that *t* teaches *c*, *s* studies *c*, and *t* is somehow eligible to be *s*'s tutor. (If it did, you would have two tutors this year on M359 and the fact that I am your tutor could be expressed via three smaller relations, "Hugh teaches M359", "<you> are studying M359", and "Hugh and <you> are in the same region".) *End of aside.*

It was E.F. Codd's great insight that gave us the notion that the way we normally record facts in databases can conveniently be represented in the form of *n*-ary relations. And he was among the first to note that *n* could even be less than 2, even though it can hardly be said that a relation of degree 1[1] is "a connection, correspondence or contrast between things". But the point I want to make is that

[1] Relations of degree 0 also exist in the theory, but alas Codd didn't notice them and so they don't appear in many texts on the subject (including M359). There are only two such relations, because there is only one 0-tuple (the tuple with no attributes and therefore no values at all). A relation of degree 0 is either empty or contains just that 0-tuple.

every relation that is of degree 2 or more does indeed represent a relationship between (or among) things.

But not all relationships (in the general sense) are represented in E-R models by relationships (in the E-R sense). Some are represented by attributes—in the E-R sense of that term, not the relational sense, though the two senses are almost identical. That an attribute represents a special case of a relationship is shown by the following alternative representation of the Nurse entity type shown in the Hospital conceptal data model:



Here each attribute has become an entity type. That NurseName is an attribute of Nursee is indicated by the IsCalled relationship; similarly, Identifies indicates that StaffNo is an attribute of Nurse. Of course, not every E-R relationship can be represented by an attribute. In fact, only those that satisfy the following conditions can be:

- The entity type to which the attribute is to belong (Nurse in my example) must have mandatory participation (indicated by a blob). That's because an attribute requires every entity of the type in question to have a value for that attribute.
- There must be no crow's foot on the entity type that is to become an attribute. That's because an attribute requires every entity of the type to which it belongs to have *exactly one* value for that attribute.
- The entity type that is to become an attribute must not participate in any other relationships. And that means it cannot have any attributes either.

The big point arising from this observation is that, really, *every* relation in a relational design is a "relation for relationship(s)"![2]

Foreign key placement and "relation for relationship"

So, what is the *real* distinction being made between the two methods—foreign key placement and "relation for relationship"—that M359 teaches for representing an E-R relationship?

Well, the distinction lies in a very important observation that comes up towards the end of Block 2 when we study *normal forms* and their relevance to database design issues. The observation is that, wherever a relational design *RD1* includes an *n*-ary relation *r* where *n* is greater than 2, it might be possible to replace *r* by two or more relations, each of degree less than *n*, such that the resulting design *RD2* is logically equivalent to *RD1* (i.e., *RD2* is capable of representing exactly the same information

[2] Codd, in his 1970 paper, used the term "relationship" to distinguish his notion of relation from the older mathematical notion in which the order in which the attributes appear has significance. Having pointed out the distinction, he then used the mathematical term anyway in the rest of the paper. Well, otherwise we wouldn't have had the useful adjective, "relational"!

as *RD1*). In such cases, we might need to consider the advantages and disadvantages of each design in deciding which one to plump for, and that is what I'll now do.

Staying with the Hospital example, consider Doctor, the entity type shown as Doctor(StaffNo, DoctorName, Position). If we ignore the connection to Specalist, we might represent it in a relational design by

relation *Doctor*
StaffNo StaffNumbers
DoctorName Names
Position Positions
primary key *StaffNo*

Here is a predicate for *Doctor*:

> The doctor with staff number *StaffNo* is called *DoctorName* and has position *Position*.

But note how that sentence uses elision to combine two sentences, logically joined together by "and", into a single sentence. If we were to write this single sentence in symbolic logic and than translate it back into English, word for word, we would arrive at

> The doctor with staff number *StaffNo* is called *DoctorName* **and** the doctor with staff number *StaffNo* has position *Position*.

Also note very carefully that as a consequence of this design, every doctor whose existence is recognised by the database *must have* a staff number, a name, and a position—exactly one of each, in fact.

Now, I can take that longwinded version of the predicate and write it as two separate sentences, replacing the word "and" by a full stop, like this:

> (a) The doctor with staff number *StaffNo* is called *DoctorName.*
>
> (b) the doctor with staff number *StaffNo* has position *Position.*

On the basis of a relation for each predicate, that would lead us to this alternative design:

relation *DoctorN*
StaffNo StaffNumbers
DoctorName Names
primary key *StaffNo*

relation *DoctorP*
StaffNo StaffNumbers
Position Positions
primary key *StaffNo*

You should be able to see easily that this design is capable of representing exactly the same information as the single *Doctor* relation. Every tuple in *Doctor* can be split into two, so long as each resulting portion includes a copy of the primary key value (*StaffNo*). We can express this observation using relational algebra, as follows:

> *DoctorN* = **project** *Doctor* **over** *StaffNo, DoctorName*
> *DoctorP* = **project** *Doctor* **over** *StaffNo, Position*
> *Doctor* = *DoctorN* **join** *DoctorP*

But if the second design is to be truly logically equivalent to the first, then it must faithfully enforce the same constraints as the first. Implicit in the first design is a constraint I have already mentioned, that every doctor whose existence is recognised by the database has exactly one each of a staff number, a name, a position and a specialism. That constraint would have to be stated explicitly in the second design, as follows:

relation *DoctorN*
StaffNo StaffNumbers
DoctorName Names
primary key *StaffNo*
foreign key *StaffNo* **references** *DoctorP*

relation *DoctorP*
StaffNo StaffNumbers
Position Positions
primary key *StaffNo*
foreign key *StaffNo* **references** *DoctorN*

So, on the assumption that the single *Doctor* relation really does meet our requirements, we would probably prefer that design to the more complicated *DoctorN/DoctorP* one (and it is actually problematical in the current state of our technology because few, if any, implementations provide adequate support for "referential cycles" like the simple one illustrated here—you would need to be able to insert tuples into both relations simultaneously to avoid violating those foreign key constraints).

But supposing the requirements turn out to have been incorrectly stated, because the assumption that every doctor has a position is not after all a safe one. In that case the following small revision of the second design is not merely preferred—*it is a necessary consequence!* All I do is remove the foreign key constraint from *DoctorN*.

relation *DoctorN*
StaffNo StaffNumbers
DoctorName Names
primary key *StaffNo*

relation *DoctorP*
StaffNo StaffNumbers
Position Positions
primary key *StaffNo*
foreign key *StaffNo* **references** *DoctorN*

Representing E-R relationships in a relational design

Now consider the relationship HeadedBy that connects Doctor entities to Team entities in the Hospital E-R model. The Team entity type is given as Team(TeamCode, TelephoneNo). We could represent it by the following relation:

relation *Team*
TeamCode TeamCodes
TelephoneNo Phonenos
primary key *TeamCode*

whose predicate is, simply

> The team with team code *TeamCode* can be contacted by phone on *TelephoneNo*.

Because a Team entity is identified by a TeamCode value, the HeadedBy relationship is necessarily represented by pairs (2-tuples) of (*Head, TeamCode*) values constituting the relation for the predicate "The team *TeamCode* is headed by the doctor *Head*". I'll now show how the two ways in which "foreign key" method results from combining this predicate with an existing one, and how the "relation for relationship" method arises from not combining it with any other predicate.

Foreign key placement method

We can use **and** to combine the "headed by" predicate with the predicate for *Team* to give

> The team with team code *TeamCode* can be contacted by phone on *TelephoneNo* **and** is headed by the doctor *Head.*

The relation for that combined predicate would be

{Design 1 - correct}
relation *Team*
TeamCode TeamCodes
TelephoneNo Phonenos
Head StaffNumbers
primary key *TeamCode*
foreign key *Head* **references** *Doctor*

Here we are using the foreign key placement method, where the *Team* relation acquires an extra attribute, *Head*, to represent the relationship. Note the inference that every team has to have a head, and in fact *exactly one* head. If that is the case (and indeed it is, if we are to believe the E-R diagram), then Design 1 is correct and could be chosen.

Alternatively, we might consider combining the "headed by" predicate with that for *Doctor*, yielding

> The doctor with staff number *StaffNo* is called *DoctorName*, has position *Position*, specialises in *Specialism*, **and** heads the team *HeadOf.*

for which the corresponding relation would be

{Design 2 - incorrect!}
relation *Doctor*
StaffNo StaffNumbers
DoctorName Names
Position Positions
HeadOf TeamCodes
primary key *StaffNo*
foreign key *HeadOf* **references** *Team*

Again we are using the foreign key method but here *Doctor* is the target of the placement. Note the inference, now, that every doctor has to be the head of some team and in fact *exactly one* team. The E-R diagram tells us that this is *not* the case, so Design 2 would be incorrect and cannot be chosen.

In case you are aware of SQL's concept of **null,** please note very carefully that no such concept forms part of relational database theory. The concept is illogical and unsound. Relational theory, however, is logical and therefore sound, which proves that **null** is not part of it! More to the point, relational theory is firmly rooted in classical logic, with its two truth values, *true* and *false*. SQL admits a third truth value, *unknown*, and that lies at the root of most of the problems introduced by **null**.

In SQL, where **null** is admitted, Design 2 can be considered, but that would not be a wise choice.

Finally, we don't *have* to combine the "headed by" predicate with either of the other two. We can just take it as it is, in which case the relation for it would be

{Design 3 - correct!}
relation *HeadedBy*
Head StaffNumbers
TeamCode TeamCodes
primary key *TeamCode*
foreign key *Head* **references** *Doctor*
foreign key *TeamCode* **references** *Team*

This is the "relation for relationship" method. Although I have marked Design 3 "correct", as things stand it is now possible not only for a doctor to exist who heads no team (as required by the E-R model) but also for a team to exist that nobody heads (contrary to the requirements of the E-R model). If we adopt Design 3, therefore, we must also add a foreign key constraint to the relation *Team*, referencing *HeadedBy*, to force every team to be headed by somebody; but we might prefer Design 1, for exactly the same reason as that which leads us to choose the *Doctor* relation in preference to the *DoctorN/DoctorP* design. Note, however, that Design 3 is rather more flexible than Design 1, in the face of possible changes in requirements. If it becomes possible for a team to have several joint heads, we would have to replace Design 1 by Design 3, with possible knock-on effects for applications. With Design 3 all we would need to do is change the primary key of *HeadedBy* to include both attributes.

1:1 relationships and alternate keys

Actually, Design 1 and Design 3 are still incomplete. Notice that they both allow several different teams to have the same head, contrary to the E-R model, whose diagram shows HeadedBy to be 1:1.

Exercise: make sure you understand fully why this is so!

To complete the design we need to add the following constraint, to Team in Design 1 and to HeadedBy in Design 3:

alternate key *Head*

An **alternate key** specification has exactly the same semantics as **primary key**.

Conclusion

In conclusion, then, we can state categorically that the relation for relationship method is never logically incorrect and is in fact required whenever any of the following conditions holds (examples are given from the Hospital conceptual data model):

- there is a circle at each end of the relationship line (e.g., Supervises);
- there is a crow's foot at each end (no examples in Hospital);
- there is a crow's foot at one end only but there is also a circle at that end (e.g., ConsistOf).

In all other cases the foreign key placement method can be used, and might even be preferred, so long as the attribute[3] in question is placed in the relation corresponding to the entity type at the end showing a blob. If there is a blob at each end, then it must be placed in the relation corresponding to the entity type at the end showing a crow's foot. If there is a blob at each end and the relationship is 1:1, then the attribute can be placed in either relation (but not both!).

END

[3] Or attributes, when composite keys are involved—it is convenient to discuss these issues in terms of singleton keys only, without any loss of generality.