Relational Database Constraints

by Hugh Darwen

One of the M359 subjects that students typically find particularly difficult is *constraints*, as expressed in both conceptual (E-R models) and in relational database designs (logical schemas). This is my attempt to explain the latter in a slightly different way to the text on the subject in Block 2, which I assume you have read. I also assume you understand the **select**, **project** and **difference** operators of the relational algebra.

A constraint is a truth-valued expression that must "always"[1] evaluate to *true.*

In order to be able to express absolutely *any* constraints that might be required, we rely on the completeness of the relational algebra. But a relational algebra expression yields a relation, not a truth value. To express a constraint, we need to be able to apply some kind of truth-valued operator—typically a comparison operator—to a relation.

It turns out that in theory we could make do with just one such comparison operator; for example: **is empty.** In other words, every constraint that might ever be needed can be written in the form *r* **is empty,** where *r* is some relational expression (possibly just a relation name).

Why do I call **is empty** a comparison operator? Because it *compares* the number of tuples in a given relation with zero. If COUNT (*r*) is the number of tuples in *r*, then *r* **is empty** is equivalent to COUNT (*r*) = 0.

Now, many constraints are naturally expressed in the form of some condition that must be *true* in every tuple of a given relation. For example, if the tuples in *r* each represent the occurrence of some process that has a starting time *st* and an ending time *et*, we probably want a constraint to the effect that in every tuple of *r* the value for *st* must be earlier than the value for *et.* If **is empty** is the only comparison operator available to us, we have to invoke a kind of double negative, remembering that if condition *c* is *true* in every tuple of *r*, then there is no tuple of *r* in which **not** *c* is *true*. Thus, we would write:

> **constraint** (**select** *r* **where not** (*st* < *et*)) **is empty**

or, perhaps more likely, the logically equivalent:

> **constraint** (**select** *r* **where** *et* ≤ *st*) **is empty**

Constraint Shorthands

Although every constraint can in theory be expressed using **is empty,** certain very commonly required constraints are very cumbersome and error-prone to write that way. Database languages typically provide useful **shorthands** to make them easier. A shorthand is typically written *inside the declaration* of some relation to which it applies.

For example, consider this one again:

> **constraint** (**select** *r* **where** *et* ≤ *st*) **is empty**

If *r* is the name of some relation declared in the logical schema, the language might usefully allow us to write this constraint, inside the declaration of *r*, in the simpler form

> **constraint** *st* < *et*

[1] i.e. whenever the database is updated

thus implying that the given expression is required to be <u>*true*</u> in every tuple of *r*.

An even commoner form of constraint that can conveniently be given as part of the declaration of a relation in the logical schema is the **primary key** constraint. This, too, is a shorthand for a certain comparison. For example, let the primary key of *r* consist of the attributes *k1* and *k2*. Then the constraint could be expressed like this:

> **constraint** COUNT (*r*) = COUNT (**project** *r* **over** *k1, k2*)[2]

If the number of tuples in *r* is the same as the number of tuples in its projection over *k1* and *k2*, then it follows that no two tuples in *r* can have the same combination of values for these two attributes—if they did, they would "condense" (so to speak) to a single tuple in the projection. The typical shorthand, imbedded in the declaration of *r*, is of course

> **primary key** (*k1, k2*)

(and the parentheses might be optional). Similarly,

> **alternate key** *k3*

imbedded in the declaration of *r*, is shorthand for

> **constraint** COUNT (*r*) = COUNT (**project** *r* **over** *k3*)

Another common form of constraint is to enforce mandatory participation of participant *p1* in some relationship between *p1* and *p2* (as expressed in the corresponding E-R model). Mandatory participation of participant *p1* typically means that every tuple in the relation representing *p1* must have at least one "matching" tuple in the relation representing *p2*. The "matching" in question is "over" some set of attributes that are considered to common to *p1* and *p2*. If that set is { *a1, a2* }, then the required constraint can be expressed like this:

> **constraint** (**project** *p1* **over** *a1, a2*) **difference** (**project** *p2* **over** *a1, a2*) **is empty**

If any tuple in *p1* has values for *a1* and *a2* such that there is no tuple in *p2* having those same values for *its a1* and *a2* attributes, then the tuple (*a1, a2*) must appear in the result of the **difference**, which is therefore non-empty. Now, in certain very special but common circumstances, the so-called **foreign key** shorthand is available to express constraints of this particular kind. The very special circumstances are as follows (and, personally, I find them to be somewhat arbitrary and over-restrictive, so I'm afraid I can't fully justify them for you):

- *p1* and *p2* are both relations that are declared in the logical schema.
- The declaration of the constraint is imbedded in the declaration of relation *p1*.
- The declaration of *p2* includes the constraint declaration **primary key** (*a1, a2*)

When these conditions all pertain, the constraint can be written as part of the declaration of *p1* like this:

> **foreign key** (*a1, a2*) **references** *p2*

END

[2] I haven't used **is empty** here because we would have to resort to a bit of trickery that I don't want to be bothered with in this short paper.