M359: Relational Databases: theory and practice
Notes to accompany slides on Constraints

Hugh Darwen

1. Cover slide

Now that we have learned relational algebra, we are in a position to tackle the two topics of this section.

As in the slides, occasional references to features of **Tutorial D** that are not included in M359 are shown in red, like this.

2. Constraints

"… the database is at all times in a *consistent* state"—we need to clarify "at all times". The normally means that consistency is enforced at the end of every innermost statement—in other words, from a syntactic point of view, every time the system encounters a semicolon in the commands that are given to it. There is no requirement for consistency to be enforced at a lower level of granularity than that, because intermediate states arising during the execution of a command are not visible to any users.

Some authorities require constraints to be satisfied only at the ends of transactions—in other words, immediately following execution of COMMIT commands. This weaker approach allows an inconsistent state to arise during a transaction and to be visible to the user running that transaction, but still protects other users.

3. KEY Constraints

These should perhaps be called *superkey* constraints. A superkey is any set of attributes that contains every attribute of some key. Thus, the uniqueness requirement for keys is satisfied, but not necessarily the *irreducibility* requirement—sometimes a superkey contains more attributes than are needed to ensure uniqueness. However, we normally assume that a key specification really does specify a key. It's just that the DBMS cannot check this, though it can of course check the uniqueness requirement.

In the example shown on the slide, the constraint, if declared, has to be satisfied for every relation ever assigned to EXAM_MARK.

Note that the current value shows that { Mark } is not a superkey, because two distinct tuples have the same value for Mark. We might wonder if { CourseId, Mark } or { StudentId, Mark } is a superkey, as both of those subsets do satisfy the constraint in the relation shown. However, our knowledge of the enterprise tells us that in general a student might get the same mark on two different courses and that different students might get the same mark on the same course.

I don't explain the complicated expression shown on the slide, as it requires too much extra knowledge of **Tutorial D**. Essentially, it says that no student can get more than one mark in the same exam—in other words, that no two distinct tuples can agree on both StudentId and CourseId.

Here's a shorter longhand in **Tutorial D** for the same constraint:

COUNT (EXAM_MARK) = COUNT (EXAM_MARK { StudentId, CourseId })

EXAM_MARK { StudentId, CourseId } is **Tutorial D**'s counterpart of **project** EXAM_MARK **over** StudetnId, CourseId. If the number of tuples in the base relvar is equal to the number of tuples in its projection over the key attributes, then the key constraint is satisfied; otherwise it is not.

4. When a Superkey Is a Key

See notes for Slide 3.

5. The key Shorthands

No notes.

6. Multiple Keys

No notes.

7. Degenerate Cases of Keys

Recall the definition of key. First we defined superkey as meaning a certain kind of uniqueness constraint, and then we defined a key as being a superkey of which no proper subset is a superkey.

As for the special property implied by the empty key, recall that if *k* is a superkey of *rv,* then COUNT(*rv*) = COUNT(*rv* { *k* }). What is the maximum value of COUNT(*rv* {})?

8. "Foreign Key" Constraints

The term "foreign key" was introduced in the 1970s and taken up by SQL. **Tutorial D** has no direct counterpart, for reasons we shall shortly see. Not everybody finds the term very intuitive. The special kind of constraint it refers to is certainly a very common one indeed, and comparatively easy to implement with good performance in a DBMS.

The constraint involves matching certain attributes (just one in the example) with those of a key of "another" relvar (called the *referenced relvar*). I wrote the word "another" in quotes, because in fact the referenced relvar is permitted to be the same as the referencing relvar; but the fact that the two relvars are usually different ones perhaps explains the use of the word "foreign". The referenced attributes are a key in a foreign place, so to speak, but unfortunately the term "foreign key" refers to the referenc*ing* attributes!

9. Inclusion Dependency

By using a relation comparison such as the one shown on this slide, we lose those restrictions that, for reasons that are not entirely clear, are imposed on foreign keys:

- The referencing relation does not have to be specifically a base relvar reference (here it is a projection of a base relvar).
- Nor does the referenced relation (here again it is a projection of a base relvar).
- And the matching attributes no longer have to constitute a declared key of the referenced relation (though here they are).

Because the reasons are unclear, the shorthand is not supported in **Tutorial D**, so in *Rel359* you will have to write such constraints using **is empty** shorthand shown on the next slide. In any case, even if **Tutorial D** did support FOREIGN KEY, it would have to be done differently from SQL, because in SQL the specification depends on column order when more than one column is involved.

Rel359 alert: Rel359 uses <= and >= for ⊆ and ⊇, respectively.

10. is empty Example

Because the operand of **is empty** is a relation expression of arbitrary complexity, and because of the completeness of the relational algebra, any constraint can be expressed using **is empty**, in theory. But sometimes the more general notation for inclusion dependencies is more convenient. The

primary key, alternate key and **foreign key** shorthands, when applicable, are always more convenient.

Note that expressions using **is empty** often seem like double negatives. We want to ensure that every mark is between 0 and 100 but we have to declare instead that no mark shall not be between 0 and 100. M359 teaches the method of declaring the constraint as a condition to be satisfied by every tuple in the relation in whose definition the constraint is declared. This gets around the need for double negatives. SQL has a similar construct.

11. Declaration of Inclusion Dependency

Note that the operands now do not have to be of the same type (have the same heading). So we don't need those projections that we had to use in the inclusion dependency. As a consequence, the expression now includes no attribute names. This makes it immune to certain changes in the database definition, but vulnerable to others. It is unaffected when the name StudentId is changed to the same new name in both relvars, but it is vulnerable to the case where it is changed in just one of them. To be 100% safe, but vulnerable to both kinds of change, the constraint could be written like this:

```
IS_EMPTY ( IS_ENROLLED_ON { StudentId } NOT MATCHING
IS_CALLED { StudentId } )
```

But use of IS_EMPTY (… NOT MATCHING …) expressions poses big performance challenges for the DBMS. Shorthands are often good for the system as well as the user. The FOREIGN KEY shorthand used in SQL does have the advantage of being easy to implement with great efficiency. The question is, do we really need *all* of the observed restrictions before we can achieve the same efficiency?

12. "Exclusion Dependency"?

"Exclusion Dependency" is in quotes because I made it up for this lecture. You won't find it in the literature.

Unlike the **foreign key** shorthand, inclusion dependency has an obvious and occasionally useful inverse, neatly captured in **Tutorial D** by omission of the word NOT.

Note that the operands of MATCHING can now be placed in either order, which is not the case with the inclusion dependency.

End of Notes