CS252:HACD Fundamentals of Relational Databases
Notes for Section 4: Relational Algebra, Principles and Part I

1. Cover slide

In this section we establish the general principles underlying the relational algebra and, in Part I, start to discover its operators.The principles lean heavily on the notion of a relation as being a representation of the extension of a predicate.The algebra is called relational because its operators operate on relations to yield relations—in mathematical parlance, its operators are *closed over* relations.

2. Anatomy of a Relation

This is a reprise of one of the slides of Section 1, Introduction.

Because of the distinction I have noted between the terms "relation" and "table", we prefer not to use the terminology of tables for the anatomical parts of a relation. We use instead the terms proposed by E.F. Codd, the researcher who first proposed relational theory as a basis for database technology, in 1969.

Try to get used to these terms. You might not find them very intuitive. Their counterparts in the tabular representation might help:

relation	table
(n-)tuple	row
attribute	column

Also (repeating what is shown in the slide):

The **degree** is the number of attributes.

The **cardinality** is the number of tuples.

The **heading** is the *set* of attributes (note set, because the attributes are not ordered in any way).

The **body** is the *set* of tuples (again, note set).

An attribute has an **attribute name**.

Each attribute has an **attribute value** in each tuple.

3. ENROLMENT Example

This is a reprise too, showing how a relation can be *depicted* as a table.

The relation shown here is a hypothetical "current" value assigned to the relvar named ENROLMENT.

Here is a possible declaration for this variable in **Tutorial D**:

```
VAR ENROLMENT BASE RELATION { StudentId SID, Name NAME, CourseId CID } ;
```

where SID, NAME and CID are the types (or domains) of the attributes StudentId, Name and CourseId, respectively. The word BASE indicates that this is a database variable rather than a local one. A database variable persists until it is explicitly destroyed.

Note the predicate for the relvar, under which each tuple in the relation assigned to it (i.e., each row in the table) is to be *interpreted*.

Because Anne is enrolled on two courses, her name is recorded twice. Not a good idea! What if one row showed Anne as the name and the other showed Ann? Or Boris?

In any case, how could we record the student identifier and name of a student who has registered with the university but is not currently enrolled on any courses? (Assuming such a strange state of affairs is permissible, that is.)

The solution is to split the ENROLMENT variable into two. The conjunction "and" in the predicate shows us how and where to make the split …

4. Splitting ENROLMENT

To be going on with, these two relvars will constitute the database for our case study. We will add more relvars when we need them.

Note the arrival of S5, called Boris, a student who is not enrolled on any courses. In the previous single-relvar design we had no means of recording S5's name while that student was not enrolled on any courses.

5. Relations and Predicates (1)

No notes.

6. Relations and Predicates (2)

Under the Open-World Assumption, it is still the case that every tuple in the relation represents a true instantiation, but it is not necessarily the case that every such tuple is included.

7. Relational Algebra

No notes.

8. Logical Operators

The operators that logicians define to operate on predicates are (a) all of those defined to operate on propositions (AND, OR, NOT) and (b) *quantifiers*.

To quantify something is to say how many of it there are. The two best known quantifiers are called EXISTS and FOR ALL, symbolically ∃ and ∀, respectively. Either of these is sufficient for all the others to be defined in terms of it, and we will take EXISTS as our primitive one. To say that some kind of thing exists is to say there there is *at least one* thing of that kind.

EXISTS is illustrated in predicate number 2 in the slide. Read it as "There exists a course CourseId such that StudentId is enrolled on CourseId." Notice how, although the variable CourseId appears twice in this rewrite, it doesn't appear at all in the shorthand used on the slide. The variable is said to be bound (by quantification). In predicates 3 and 4 we have replaced the variable Name by the names Devinder and Boris. That variable is also said to be bound, but by *substitution* rather than by quantification.

Variables that are not bound are called *free variables*, or *parameters* (because a predicate can be thought of as an operator that when invoked yields a truth value).

AND, OR, and NOT are illustrated in predicates 1, 4 (which also uses NOT) and 3 (which also uses AND), respectively.

9. Meet The Operators

The left-hand column shows the familiar operators of predicate logic. The right-hand column gives names of corresponding relational operators. These relational operators constitute the relational algebra.

Why so many for AND? We will see that JOIN is the fundamental one, but we need others for convenience, to cater for various common special cases that would be too difficult if JOIN were our only counterpart of AND.

Why some in capitals and others in lower case? The ones in caps are operator names that are actually used (normally) in concrete syntax for invoking the operators in question; the other operators are (normally) invoked using different notation, as we shall see. In particular, the ones in caps are used that way in **Tutorial D**.

Advance warning: When we come to the treatment of OR and NOT we will find that relational algebra imposes certain restrictions to overcome certain severe computational problems with these operators. For this reason, relational algebra is not complete with respect to logic (first order predicate calculus, to be precise). Instead we will make do with what E.F. Codd called *relational completeness*.

10. JOIN (= AND)

Note the format of Slide 10 carefully. The format is used for other examples too. The top line shows a predicate for which we provide a corresponding relational expression. The next line, with upward arrows connecting its parts to part of the predicate, is that relational expression—in this case an invocation of the operator `JOIN`. Underneath that is a table depicting the relation resulting from that invocation.

Note that `JOIN` here is an *infix* operator, placed in between its operands, like the usual arithmetic operators.

The relational expression corresponding to the given predicate is
`IS_CALLED JOIN IS_ENROLLED_ON`.
We can also write it as:
`IS_ENROLLED_ON JOIN IS_CALLED`, or
`JOIN { IS_ENROLLED_ON, IS_CALLED }`, or
`JOIN { IS_CALLED, IS_ENROLLED_ON }`.

Those last two examples show `JOIN` being used as a *prefix* operator, written in front of the operands, the operands being enclosed in braces (rather than parentheses) to indicate that the order is unimportant.

The upwards arrows show which bits of the relational expression correspond to which bits of the predicate.

The arrows connecting rows in the tables show which combinations of operand tuples represent true instantiations of the predicate. Note how a tuple on the left "matches" a tuple on the right if the two tuples have the same StudentId value; otherwise they do not match.

The result of this `JOIN` is shown on the next slide …

11. IS_CALLED JOIN IS_ENROLLED_ON

Note very carefully that the result has only one StudentId attribute, even though the corresponding variable appears twice in the predicate. Multiple appearances of the same variable in a predicate are always taken to stand for the same thing. Here, if we substitute S1 for one of the StudentIds, we must also substitute S1 for the other. That is why we have only one StudentId in the resulting

relation. StudentId is called a *common attribute* (of *r1* and *r2*). In general, there can be any number of common attributes, including none at all.

12. Definition of JOIN

Commutative means that the order of operands is immaterial: *r1* JOIN *r2* ≡ *r2* JOIN *r1*.

Associative means that (*r1* JOIN *r2*) JOIN *r3* ≡ *r1* JOIN (*r2* JOIN *r3*).

From the given definition we can conclude:

(a) Each attribute that is common to both *r1* and *r2* appears once in the heading of the result.

(b) It is only where *t1* and *t2* have equal values for each common attribute that their combination (via union) yields a tuple of the result. If *t1* and *t2* have different values for common attribute *c,* then their union will have two distinct *c* attributes and therefore is not a tuple and cannot appear in the result.

(c) If there are no common attributes, then the heading of the result consists of each attribute of *r1* and each attribute of *r2*. It follows that every combination of a *t1* with a *t2* appears in the result.

(d) If either operand is empty, then so is the result.

How would we achieve the same natural join, of IS_CALLED and IS_ENROLLED on, if the student identifier attributes had different names? We must be able to do that, or we lose relational completeness! We introduce a relational operator that has no direct counterpart in predicate logic: RENAME.

13. RENAME

Unfortunately, E.F. Codd did not foresee the need for a RENAME operator and so it was omitted from some prototype implementations of the relational algebra. And there is no counterpart of RENAME in SQL.

RENAME returns its input unchanged apart from the specified change(s) in attribute name.

In **Tutorial D**, multiple changes can be specified, separated by commas. For example:

RENAME IS_CALLED (StudentId AS Id, Name AS StudentName)

14. Definition of RENAME

No notes.

15. RENAME and JOIN

Each operand of the JOIN is an invocation of RENAME.

It is perhaps a trifling irritating to be told that each of our students has the same name as himself or herself. (Note in passing that "*x* has the same name as *y*" is an example of what is called a *reflexive* relation—true whenever *x*=*y*.) Soon we will discover how those truisms can be eliminated from the result.

It is also a trifle annoying to be told not only that S2 has the same name as S5 but also that S5 has the same name as S2.(Note in passing that "*x* has the same name as *y*" is an example of what is called a *symmetric* relation—if it is true for *x*=*a* and *y*=*b*, then it is true for *x*=*b* and *y*=*a*.) We will discover how that truism can be eliminated, too.

16. Special Cases of JOIN

With some of the relational operators we can take note of certain special cases of their invocation, just as we do with operators in other algebras. For example, in arithmetic, we note that adding 0 to any number *x* is a special case of addition because it always yields *x* itself. Similarly, "times 1" is a special case of multiplication. Each of these two examples involves the so-called *identity* under the operator in question: 0 for addition and 1 for multiplication. With some dyadic operators the case where the two operands are equal is also an interesting special case. For example, *x*-*x* is always equal to 0, and *x*/*x* is always equal to 1 (except when *x*=0 of course, when it is undefined).

In set theory the empty set (usually denoted by the greek letter phi: ϕ) is such that for an arbitrary set *A*, $A \cup \phi = A$, $A - \phi = A$, and $A \cap \phi = \phi$. We can therefore note these three cases as special cases of union, difference, and intersection, respectively.

In *R* JOIN *R*, all attributes are common to both operands and each tuple in *R* "matches" itself and no other tuples. Therefore the result is *R*.

When *r1* and *r2* have the same heading, then the result of *r1* JOIN *r2* consists of every tuple that is in both *r1* and *r2*. Traditionally, such cases are allowed to be written as *r1* **INTERSECT** *r2*, and **Tutorial D** allows that. It could be argued that there isn't much point, but we will see some justification (not much, perhaps) when we eventually come to relational UNION. Recall how in set theory, intersection corresponds to AND whereas union corresponds to OR.

When *r1* and *r2* have disjoint headings (i.e., no attributes in common), then every tuple in *r1* matches every tuple in *r2* and we call the join a Cartesian product. Some authorities permit or require *r1* TIMES *r2* to be written in this special case. Permitting it is perhaps a good idea, for the user is thereby confirming that he or she is fully aware of the fact that there are no common attributes in the operands. But *requiring* the use of TIMES in this special case turns out to be not such a good idea. Exercise: Why not? (Hint: look at the notes for Slide 12)

17. Interesting Properties of JOIN

Because of these properties, **Tutorial D** allows JOIN to be written in prefix notation, with any number of arguments:

JOIN { *r1, r2, ... rn* }

18. Projection (= EXISTS)

We say that IS_ENROLLED_ON is *projected over* StudentId.

We can project a relation over *any* subset of its attributes. The chosen subset gives us the heading of the result.

Note, however, that we are *quantifying* over the excluded attribute(s), in this case CourseId. It is considered important to be able to write either the attributes to be included or those to be excluded, whichever suits best at the time. Accordingly, Tutorial D permits the present example to be written thus: IS_ENROLLED_ON { ALL BUT CourseId }.

In fact, **Tutorial D** supports the ALL BUT notation *everywhere* that an attribute name list can appear.

19. Definition of Projection

No notes.

20. How ENROLMENT Was Split

When, as in the examples on the slide, the declared type for the variable can be inferred from the expression denoting the operand of INIT[1], you can even avoid the tedious repetition of ENROLMENT { StudentId, Name }, like this:

```
VAR IS_CALLED BASE
        INIT (ENROLMENT { StudentId, Name })
        KEY { StudentId } ;
```

21. Special Cases of Projection

The first example is the projection of R over all of its attributes; the second is its projection over no attributes.

In spite of their names, TABLE_DEE and TABLE_DUM are the only two relations that cannot be depicted as tables! This is because they have no attributes, so the corresponding tables, having no columns, would be invisible.

22. Another Special Case of JOIN

No notes.

End of Notes

[1] The cases where it cannot necessarily be inferred arise when subtyping is used, but that optional feature of **Tutorial D** is beyond the scope of CS252.