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

Hugh Darwen

1. Cover slide

An algebra is a set of operators that somebody has found interesting enough to describe, note their properties, and (probably) propose them for some particular purpose. Arithmetic is one such set (plus, minus, times, and divide). Propositional logic (AND, OR, NOT, etc.) is another.

In this lecture I first establish the general principles underlying the relational algebra and then describe the operators, using simple examples.The principles lean heavily on the notion in logic 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.

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

2. Anatomy of a Relation

This is a reprise of one of the slides of my tutorial 1 lecture, 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 the notation used by M359 in Block 2:

```
relation ENROLMENT
  StudentId SID
  Name NAME
  CourseId CID
  primary key (StudentId, CourseId) ;
```

where SID, NAME and CID are the domains (nowadays normally referred to as *types*) of the attributes StudentId, Name and CourseId, respectively.

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 <u>CourseId</u> such that <u>StudentId</u> is enrolled on <u>CourseId</u>." Notice how, although the variable <u>CourseId</u> 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 (in some languages, such as **Tutorial D**, but not M359)
`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 <u>StudentId</u> 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 <u>StudentId</u> 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 <u>StudentIds</u>, we must also substitute S1 for the other. That is why we have only one <u>StudentId</u> 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.

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*, as taught in M359. 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*, as taught in M359, 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** (but not M359!) 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** (but not M359!)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

In Block 2 M359 does not teach any operators for database updating. This slide uses the **Tutorial D** assignment operator (:=) to assign the result of relational projection to each of the two variables, IS_CALLED and IS_ENROLLED on into which we split the original ENROLMENT variable.

21. Special Case of AND (1)

The very special case here is a simple substitution of a value, 'Boris', for one of the predicate variables, yielding a 1-place predicate. The corresponding relation therefore has just a single attribute, StudentId. We achieve this by a combination of *restriction* (the new operator, **where**, introduced on this slide) and projection.

It is the restriction that is a relational counterpart of AND. The predicate for the entire expression, as shown on the slide, is

There exists a *Name* such that *StudentId* is called *Name* and *Name* is Boris.

The projection over *StudentId* corresponds to the "There exists a *Name* such that" part of the predicate. The text following "such that" is the predicate for the **where** invocation:

StudentId is called *Name* and *Name* is Boris.

In general, the expression that appears after the key word **where** is a *conditional* expression, typically involving comparisons, possibly combined using the usual logical operators, AND, OR and NOT.

Note that the conditional expression can, and typically does, reference attributes of the input relation.

22. A More Useful Restriction

In this slide we are applying the restriction "**where** sid1 < sid2" to the **join** of two invocations of **rename**. The parentheses make that order of operations clear if you study them very carefully, but although they are easily understood by the computer, they are something of a burden to the human reader.

Note how that **where** invocation cannot conveniently be expressed by a **join**, as we did in Slide 3, with a relation of degree 1 and cardinality 1, to obtain student ids of students called Boris. If we tried a similar technique here, we would have to join with a binary relation giving all pairs of student ids where the first compares less than the second. When would we ever stop writing?

Now we know why **where** is a *very* useful operator for certain (very common) special cases of predicates involving AND.

23. Definition of Restriction

"On attributes of *r*" simply means that *c* can contain zero or more references to attributes of *r*.If *c* contains no references to attributes of *r*, then its result (TRUE or FALSE) is the same for each tuple of *r*.

24. Special Cases of Restriction

No notes.

25. Special Case of AND (2)

Sorry about the contrived example. I could have given a more realistic one if my database had some numerical data in it. For example, think about computing the price of each item in an order. That would involve multiplying the unit price by the quantity ordered and perhaps applying a discount agreed for the customer placing the order.

26. Extension

Unfortunately, E.F. Codd did not foresee the need for an EXTEND operator and so it was omitted from some prototype implementations of the relational algebra and some textbooks still fail to mention it.

In this example, we use the operator, SUBSTRING. SUBSTRING(*s*, *b*, *n*) returns the string consisting of the *n* characters of *s* beginning at position *b*. Thus, SUBSTRING(Name, 1, 1) yields the string consisting of the first character of the Name value in each tuple of IS_CALLED.

The relation corresponding to the SUBSTRING operator contains a 4-tuple for every possible combination of *s*, *b*, and *n* values (compare with the relation for arithmetic "plus", shown in my lecture on Constraints, Slide 6). To write this relation out in full would take forever (nearly), and that's why the equivalent expression using **join** is impractical.

27. Definition of Extension

Exercise:

Assume the existence of the following relvars:

CUST with attributes <u>C#</u> and DISCOUNT

ORDER with attributes <u>O#</u>, C#, and DATE

ORDER_ITEM with attributes <u>O#, P#,</u> and QTY

PRODUCT with attributes <u>P#</u> and UNIT_PRICE

The underlined attributes are those specified in a **primary key** declaration for each relvar. Thus, for example, there cannot be more than one order item for the same part in the same order.

The price of an order item can be calculated by the formula QTY*UNIT_PRICE*(1-(DISCOUNT/100)).

Write down a relation expression to yield a relation with attributes O#, P#, and PRICE, giving the price of each order item.

28. OR

The dotted line indicates that the table depicting the relation that would represent the extension of that predicate is incomplete; for the relation, under our Closed-World Assumption, must include every tuple that satisfies *either* of the two *disjuncts* (the sentences connect by “or”: *StudentId* is called *Name*, and *StudentId* is enrolled on *CourseId*).

It is neither reasonable nor very practical to require the DBMS to support evaluation of such huge relations, so Codd sought some restricted support for disjunction that would be sufficient to meet the perceived practical requirement …

29. union (restricted OR)

As it happens, analogous restrictions are typically found in other logic-based languages, such as Prolog. By enforcing the operands to be *type compatible* (i.e., have the same heading), the combinatorial explosion depicted on the previous slide is avoided!

30. Definition of union

Recall that *r1* **intersect** *r2* is traditionally permitted in the special case where the heading of the operands are identical. In basic set theory, we have union, intersection, and difference. It seems that Codd thought his relational algebra would seem psychologically incomplete unless it had a counterpart of each of those three.

Exercises:

1. What is the result of *r* **union** *r*?
2. Is **union** commutative? I.e., do *r1* **union** *r2* and *r2* **union** *r1* always denote the same relation?
3. Is **union** associative? I.e., do (*r1* **union** *r2*) **union** *r3* and *r1* **union** (*r2* **union** *r3*) always denote the same relation?

31. NOT

Again the Closed-World Assumption makes general support for negation a no-no (pun intended!).

Codd’s solution was the same as with disjunction, with his definition of **minus** (see later), but after Codd a significantly less restrictive approach was discovered …

32. Restricted NOT

In **Tutorial D** (but not M359!) we define an operator, NOT MATCHING, that combines negation with *conjunction*, thus avoiding the combinatorial explosion. Again, analogous restrictions are found in other logic-based languages.

33. Definition of NOT MATCHING

Exercises: State the result of

1. *r* NOT MATCHING *r*
2. (*r* NOT MATCHING *r*) NOT MATCHING *r*
3. *r* NOT MATCHING (*r* NOT MATCHING *r*)

Is NOT MATCHING associative? Is it commutative?

34. difference

M359 teaches **difference,** which completes Codd’s trio of counterparts of the basic set operators.

To define **difference** in terms of NOT MATCHING is trivial, for *r1* **difference** *r2,* for all the cases where it is defined, is equivalent to *r1* NOT MATCHING *r2*.

Exercise: Define *r1* NOT MATCHING *r2* in terms of **difference**.

35. Example of difference

Note how **difference** so often has to be used in conjunction with **project** and **join**, if one's query is to be useful. This example is equivalent to the **Tutorial D** expression IS_CALLED NOT MATCHING IS_ENROLLED_ON.

End of Notes