CS252:HACD Fundamentals of Relational Databases
Notes for Section 5: Relational Algebra Part II

1. Cover slide

In Part I we met JOIN and projection, the relational algebra counterparts of AND and existential quantification, respectively. We also met RENAME.In this section we look at two more counterparts of AND, dealing with certain special cases that need special treatment. These are called *restriction* and *extension.* Then we look at *aggregation,* a rather more complicated process that also relates to AND.Finally, we move on to counterparts of the other two logical operators, OR and NOT.

2. The Running Example

No notes.

3. Special Case of AND (1)

The very special case here is a simple substitution of a value, NAME ('Anne'), 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.

Alert regarding *Rel*: If you try this example in *Rel*, you will have to write just 'Boris' in place of NAME('Boris') if the attribute Name is of type CHAR.

4. 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. Here is an alternative formulation, using WITH to avoid the use of parens:

```
WITH IS_CALLED RENAME ( StudentId AS Sid1 ) AS t1,
     IS_CALLED RENAME ( StudentId AS Sid2 ) AS t2,
     t1 JOIN t2 AS t3,
     t3 WHERE Sid1 < Sid2 AS t4 :
t4 { Sid1, Sid2 }
```

WITH is a mechanism for *introducing names* into an expression, allowing the *introduced names* to be used *after* they been defined. (If they were used before they are defined, we would have recursion, a matter that I will not discuss further in CS252.) The list of name introductions, separated from each other by commas, ends with a colon. After the colon comes the expression defining the required final result.

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 shorthand for certain special cases of predicates involving AND.

5. 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*.

6. Special Cases of Restriction

No notes.

7. 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.

8. 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 defined in the script OperatorsChar.d provided by *Rel* in its directory named Scripts. SUBSTRING(*s*, *b*, *n*) returns the string consisting of the *n* characters of *s* beginning at position *b* (with 0 being the first position in *s*). Thus, SUBSTRING(Name, 0, 1) yields the string consisting of the first character of the Name value in each tuple of IS_CALLED. Note carefully that the expression assumes, possibly wrongly, that SUBSTRING is defined for values of type NAME as well as for values of type CHAR. If NAME is defined in like manner to that shown in Lecture HACD.2, Slides 14 and 15, then we would have to replace "Name" in the invocation by "THE_C(Name)", meaning "the value of the component C of the declared possible representation (possrep) for type NAME".

The binary 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 Lecture HACD.2, "Values, Types, Variables, Operators", Slide 4). To write this relation out in full would take forever (nearly), and that's why the equivalent expression using JOIN is impractical.

9. Definition of Extension

Exercise:

Assume the existence of the following relvars:

CUST with attributes C# and DISCOUNT

ORDER with attributes O#, C#, and DATE

ORDER_ITEM with attributes O#, P#, and QTY

PRODUCT with attributes P# and UNIT_PRICE

The underlined attributes are those specified in a 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.

10. Two More Relvars

Exercise:

> Write down a relational expression to give, for each pair of students sitting the same exam, the absolute value of the difference between their marks. Assume you can write ABS(x) to obtain the absolute value of *x*.

But I am introducing exam marks to illustrate our next subject, aggregation …

11. Relations within a Relation

Here I'm only showing that such relations do exist. We'll see the real significance of this one shortly. A possible predicate for this one is "Exam_Result gives the marks obtained by all the students who took the exam for course CourseId."

Note that the information content of C_ER is identical to the combined information content of COURSE and EXAM_MARKS.

Relations that are attribute values are sometimes referred to as *nested relations*.

C_ER can be obtained, using relational operators we have already learned, from the COURSE and EXAM_MARKS relvars just shown …

12. To obtain C_ER from COURSE and EXAM_MARK:

Here the extension formula for the added attribute Exam_Result is a relation expression.

Note how the JOIN in the Exam_Result formula yields a relation consisting of just those tuples in EXAM_MARK that match the CourseId of the tuple on which the formula is being evaluated. Note also how this yields an empty relation for course C4.

"CourseId CourseId" looks a bit funny, especially in black and white. In the real slide the first CourseId appears in blue, indicating an attribute name. The second appears in black, indicating an expression denoting a value. So "CourseId CourseId" specifies that the value for the CourseId attribute of the tuple is to be the value of the CourseId attribute taken from each tuple in turn of COURSE.

The projection over {ALL BUT CourseId } yields the binary relations you see nested inside C_ER. In the next lecture you learn the **Tutorial D** "shorthand", COMPOSE, which combines JOIN and projection. If I replace the word JOIN by COMPOSE in this slide, the projection over {ALL BUT CourseId} can be omitted. COMPOSE does a join and then projects the result over the noncommon attributes only.

The final projection merely discards the Title attribute to give the result as shown on the previous slide.

13. An Aggregate Operator

COUNT (IS_ENROLLED_ON) is an invocation of COUNT. In general, the operand of COUNT is any relation *r* and COUNT (*r*) gives the cardinality of *r*.

I use "=" here just to show you the result of the invocation when it is applied to the value of IS_ENROLLED_ON that we are running with.

14. More Aggregate Operators

These should all be self-explanatory, but note that, unlike COUNT, these all have a second parameter, an expression denoting the values, obtained from the tuples of the first operand, whose sum, average, maximum value, or minimum value is to be computed.

15. Nested Relations and Agg Ops

Since C_ER doesn't exist as a relvar but rather has to be derived by the expression given earlier, it would be rather nice to have a shorthand to operate on EXAM_MARK and do the whole job.

Enter **SUMMARIZE** …

16. SUMMARIZE BY

This is pretty useful, but what if we want the result to include information about exams that nobody took? In that case we need an operator that can be used to bring COURSE into the picture, too …

17. SUMMARIZE PER

SUMMARIZE PER raises the question of what to do about aggregation over the empty set.

COUNT and SUM are both easy. The sum of no numbers is zero, and so is their count. Their average, alas, has to be undefined, because the average is the sum divided by the count and division by zero is undefined.

The maximum of no numbers appears to be undefined, but given that our types are finite, there is some number that is the lowest one representable (i.e., the lowest member of the type)—some very large negative number. In theory, that could be taken as the maximum of no numbers, as I now try to explain.

SUM is repeated addition. The so-called *identity* under addition—that number that when added to any number *n* yields *n* itself, is zero. To determine the result of applying some operator "repeatedly" to the empty set, we take the identity under that operation if one exists.

MAX is repeated "take the higher of". The identity under "take the higher of" is that number that is higher than no number.

By a similar argument, the minimum of no numbers is the highest number of the type in question.

18. 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* (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 …

19. 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!

20. 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?

21. 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 …

22. Restricted NOT

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.

23. Definition of NOT MATCHING

Exercises: State the result of

1. *r* NOT MATCHING TABLE_DEE
2. *r* NOT MATCHING TABLE_DUM
3. *r* NOT MATCHING *r*
4. (*r* NOT MATCHING *r*) NOT MATCHING *r*
5. *r* NOT MATCHING (*r* NOT MATCHING *r*)

Is NOT MATCHING associative? Is it commutative?

24. MINUS

MINUS completes Codd's trio of counterparts of the basic set operators. You might like to use it in place of NOT MATCHING when it is appropriate to do so. If it turns out not be appropriate, you get a syntax error, and that might be helpful to you if it reveals that you were under some false assumption. A similar comment could be made in favour of using INTERSECT instead of JOIN, when appropriate.

To define MINUS in terms of NOT MATCHING is trivial, for *r1* MINUS *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 MINUS.

End of Notes