CS319: Relational Algebra

(revisited, reviewed, revised, simplified)

1. Cover slide

You were introduced to relational algebra in CS258. Hugh Darwen's invited CS319 lectures use a different notation and in some important respects different semantics from what you were taught in CS258, which was based on a somewhat flawed definition still appearing in some textbooks.

The principle underpinning relational algebra is 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.

Note very carefully: Relational algebra, of itself, has nothing to do with databases! And its operators are mathematical ones: they have nothing to do with updating. However, the motivation for them is of course their perceived applicability to many database problems.

Relational algebra is not a language either, but we do need some kind of notation for defining and describing its operators. The notation and semantics described here are from **Tutorial D**, a complete programming language devised for teaching purposes by C.J. (Chris) Date and the present author, first published in 1998. *Rel* is a full implementation of **Tutorial D** developed by Dave Voorhis of the University of Derby. It is available as Open Source software at http://dbappbuilder.sourceforge.net. The **Tutorial D** notation is used in the present author's free download textbook, *An Introduction to Relational Database Theory*, available at http://bookboon.com/en/an-introduction-to-relational-database-theory-ebook.

Tutorial D also pervades all of Chris Date's many books written since the mid-1990s. For the best part of 40 years Date has been far and away the most prolific (and most authoritative, and most approachable) writer on the subject of relational databases, which he started to teach hot on the heels of Codd's 1970 paper.

All of that said, however, I must emphasise that notation is relatively unimportant, so long as it's sound, intelligible and economical. It's the *semantics* that really matter.

2. Anatomy of a Relation

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).

An attribute has an **attribute name** and is of some **type** (Codd's "domain").

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

Each attribute has an **attribute value**—a value of the applicable type—in each tuple.

The terms heading and body help us to provide concise definitions of relational operators, as shown on some of my slides.

3. Running Examples

Notice the names above the two tables in this slide: IS_CALLED and IS_ENROLLED_ON and recall that no such name is shown on the previous slide. A relation doesn't have a name as such: it is just a heading and a body. IS_CALLED and IS_ENROLLED_ON, contrary to what you sometimes see in the literature, are not relation names. They are names of *variables*—relation variables, to be precise, and possibly *persistent* variables that form part of a relational database (where relation variables—*relvars* for short—are the only kind permitted).

To be going on with, these two examples will constitute the database for our case study. We will add more when we need them. The operators of the algebra operate on relations, not relation variables, but variable names are a convenient way of denoting relations appearing as operands: the name denotes the value currently assigned to the variable in question—a concept that you will of course be very familiar with if you've done any computer programming at all.

Notice also the sentences below the tables. These sentences denote the *predicates* that tell us how these relations are to be interpreted, as explained in the next two slides. Knowing the exact meanings of relations is essential, of course—otherwise we have data but no information and the relations are useless.

4. Relations and Predicates (1)

The distinction between the extension of a predicate (as a set of propositions) and a relation representing such an extension is so slight that some texts ignore it altogether and refer to the relation as the extension. However, you obviously can't see what is meant by a tuple just by "reading" it. Moreover, the same tuple might appear in two or more distinct relations, with different meanings, just as the ordered triple <2,2,4> might mean either 2+2=4 or 2×2=4 (or something else altogether), for example.

5. Relations and Predicates (2)

Under the alternative *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. If relational databases were based on the Open World Assumption they wouldn't be very useful, as we could conclude nothing from the absence of a tuple.

6. Relational Algebra

To repeat from my notes on the cover slide relational algebra, contrary to some texts on the subject, is not a language, any more than arithmetic is a language. Also contrary to some texts, it is not "procedural" (unless you think arithmetic is procedural). It is just a set of operators, but of course, as I also wrote in my notes on the cover slide, we do need some notation for the purposes of defining and using those operators, just as we need notation such as $(x + 10)/y$ for arithmetical purposes.

And, by the way, the term is *operator*, not "operation" as some writers have it.

Although being closed over relations means what it says: the result of invoking such an operator must *be* a relation, so it must have a heading in which every attribute is uniquely identified by its name. Moreover, the operator's definition must not depend on some hypothetic ordering to the attributes of its operand relation(s). I mention those two points specifically because they are not properly adhered to in some accounts you may find in the literature.

7. Logical Operators

The operators that logicians define to operate on predicates are (a) all of those defined to operate on propositions, such as AND, OR, and 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 other quantifiers 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 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.

8. Relational 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 lower case are abstract nouns used in normal text. The ones in caps refer to operators for which no such abstract noun has been appeared in the literature; these are verbs commonly used in concrete syntax for invoking the operators in question—in particular, they are used that way in **Tutorial D**. Some texts use Greek and mathematical symbols, most of which were used by Codd in 1970, and these are shown too. There are no such symbols for extension and summarization because Codd didn't define these operators.

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 make do with what E.F. Codd called *relational completeness* and we will find that the restrictions are of no concern at all in a language for computation.

9. A Bit of History

1970, E.F. Codd:

> Codd was rightly hailed for his landmark papers but unfortunately he showed little interest in the design of computer languages, in particular the new languages that would be needed to bring his ideas to fruition. I will explain the flaws and incompleteness of his account of relational algebra in my notes on later slides. Here let it just be said that for those (such as myself) who were immediately interested in developing a relational DBMS his account raised several questions of clarification to which he did not give satisfactory answers.

1975, Hall, Hitchcock, Todd:

> These authors were members of a research team working at IBM's UK Scientific Centre at Peterlee in the north-east of England. They devised a language named ISBL (Information Systems Base Language), which satisfactorily answered those questions and filled in the gaps. With one exception, all of the operators described in this lecture were directly supported in ISBL, using single character symbols available on normal keyboards, such as *, #, $.
>
> When ISBL was implemented by the same team as a prototype relational DBMS, the IBM bosses wanted to downplay the significance of this work because they feared that customers of IBM's prerelational database systems would read too much into it and expect the new kind of DBMS to be commercially available right away. Thus, the name of the prototype was PRTV, standing for Peterlee Relational Test Vehicle.
>
> Unfortunately, when the new breed of DBMS did later become available, IBM went for some different work that had been done by a bigger and much more visible team in America—work that ignored ISBL and gave rise to a very badly designed language called SQL, the subject of my final contribution to CS319, *The Askew Wall.*

1998, Date and Darwen:

> In 1994 Date and I had published a paper, *The Third Manifesto*, attempting to lay down the law, so to speak, as to what a correct implementation of Codd's relational model should look like. It was partly inspired by the many mistakes that had been made in SQL. The paper was a bullet list of prescriptions (what must be done) and proscriptions (what must *not* be done), devoid of explanation and illustration. When we came to work on a book to explain and illustrate each of these prescriptions and proscriptions, we realised that we needed to devise a complete conforming language. We hoped this language would be used for teaching purposes, hence its name, **Tutorial D** (always written in bold).
>
> Happily, the language caught the attention of Dave Voorhis and he decided to develop *Rel* (always written in italics), an implementation of **Tutorial D**, using Java. A subset of **Tutorial D** was formerly taught by Hugh Darwen in the second year module CS252, discontinued in 2012 when it was superseded by CS258. His book mentioned in the notes on the cover slide is based on his CS252 lectures.

2011, Elmasri and Navathe:

> Elmasri and Navathe's slides are the ones used for teaching relational algebra in CS258. Unfortunately they perpetuate many of the errors of the past.

10. JOIN (= AND)

Note the format of this slide 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 uses *infix* notation, the operator name being placed in between its operands, as with 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 in *prefix* notation, the name being 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 *StudentId*s, 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.

`JOIN` is the operator that Codd called *natural join*, but unfortunately he didn't regard it as primitive. Instead he chose Cartesian product as his primitive operator corresponding to AND, but his definition of Cartesian product was flawed. For example, if that operator were to be applied to the two relations shown on this slide, it would result in a "relation" with two attributes having the same name, and such an object is not a relation! The inventors of ISBL correctly chose natural join is the preferred primitive, noting that it becomes Cartesian product (without flaws) in the special case where the operands have no common attributes.

12. Definition of JOIN

Commutative means that the order of operands is immaterial: $r1$ JOIN $r2 \equiv r2$ JOIN $r1$.

Associative means that ($r1$ JOIN $r2$) JOIN $r3 \equiv r1$ 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 an attribute renaming operator. That need was first observed by the inventors of ISBL, but others appear to have independently recognized it since and some texts, including Elmasri and Navathe's, describe a version that doesn't stand up to close scrutiny. The operator described here is **Tutorial D**'s equivalent of the ISBL operator. 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

The definition needs no further elaboration but it should be contrasted with the corresponding operator offered by Elmasri and Navathe:

$$\rho_{x(A_1, A_2, \ldots, A_n)}(r)$$

where:

- r is a relation of degree n.
- ($A_1, A_2, \ldots, A_n$) is a list of attribute names, which can be omitted
- x is a "relation name", which can be omitted

Let s denote the result of an invocation of ρ.

If a list of attribute names is provided, then A_1 becomes the name of the "first" attribute of s, A_2 the name of the "second", and so on. Thus the definition depends on a hypothetical ordering to the attributes of r, but by definition the attributes of a relation are unordered!

If a "relation name" x is provided, then the semantics are not made clear but it seems from its use in examples that it allows each attribute of s to have more than one name, such that the attribute A_1 can be referred to as either A_1 or $x.A_1$. It is claimed that the dot-qualified name allows Codd's flawed version of Cartesian product to be supported, as illustrated in the following example.

Consider relations *r1* with attributes A and B and *r2* with attributes B and C. Then the Cartesian product $r1 \times r2$ denotes a relation-like object having four attributes, two of which are both named B. Clearly, any subsequent references to attribute B are ambiguous and the "relation name" operand is proposed to solve that problem: the Cartesian product $r1 \times \rho_x(r2)$ denotes a relation with attributes A, B, *x*.B, and *x*.C, with the additional proviso that *x*.C can be referred to unambiguously by its simple name C. Furthermore, these authors seem to assume that if *r* and *r2* are names of relation variables, say R and S, then those names can also be used as prefixes, such that attribute A is also named R.A. In that case the attributes of R × S can be referred to as R.A, R.B, S.B, and S.C. The nonsense of all this begins to emerge when you consider the use of an invocation of ρ as an operand, for example, as an operand of another invocation of ρ. Let $s = \rho_y(\rho_x(r))$. If r has an attribute named A, what is or are the name(s) of the corresponding attribute in s? Possibilities to be considered are A, x.A, y.A, and y.x.A (though presumably not x.y.A).

If you found the discussion of Elmasri and Nathe's ρ difficult to follow, just reflect on the wisdom of those earlier but overlooked members of the ISBL team.

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 the relation for "*x* has the same name as *y*" is an example of what is called a *reflexive* relation—the predicate is 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 the relation for "*x* has the same name as *y*" is an example of what is called a *symmetric* relation—if the predicate 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 sometimes 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 writers 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 on* {StudentId}, or *projected over* {StudentId} (take your choice).

We can project a relation on *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 better 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 in an expression.

19. Definition of Projection

No notes.

20. 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. Unfortunately Codd failed to recognize their existence and so does most of the literature on the relational model, even though they *were* recognized by the ISBL team.

21. Special Case of AND (1)

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.

22. 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. Needless to say, it was the ISBL team who came up with it and the operator described here is theirs under a different name (ISBL uses # as the operator name).

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 predicate "*Name* begins with *Initial*" contains a 2-tuple for every value of type CHAR paired with its first character. To write this relation out in full would take forever (nearly), and that's why an equivalent expression using JOIN is impractical.

23. Definition of Extension

Now that we have defined extension, we can revisit RENAME and define it in terms of extension and projection: *r* RENAME {*A* AS *B*} is equivalent to (EXTEND *r* : {*B* := *A*)}{ALL BUT *A*}.

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.

24. Special Case of AND (2)

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

Beware! Codd's name for this operator was SELECT. It was changed to restriction when SQL came along in 1979 and injudiciously used the key word SELECT for a different purpose altogether. Unfortunately not all the books have yet (2013) caught up with this change.

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.

Now, the notes for Slide 15 promise a solution to the problems arising from reflexivity and symmetry. Let *s* be the relation shown in Slide 15. Then *s* WHERE NOT(Sid1 = Sid2) eliminates the cases where students are shown as having the same name as themselves. Moreover, if we can assume that values of type SID (the type of the attributes Sid1 and Sid2) are ordered so that the operator < is defined for this type, then *s* WHERE Sid1 < Sid2 eliminates the redundant cases arising from symmetry as well as those arising from reflexivity.

Note how *s* WHERE Sid1 < Sid2 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 many cases of predicates involving AND.

25. 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*.

As for the alternative definition mentioned in the slide, `r WHERE c` is equivalent to

```
((EXTEND r : { X := c }) JOIN RELATION{TUPLE{X TRUE}}){ALL BUT X}
```

Explanation:

1. `(EXTEND r : { X := c })`, where *X* is not an attribute name of *r*, extends *r* with an attribute *X* of type BOOLEAN, giving the result of evaluating the condition *c* for each tuple of *r*.
2. `RELATION{TUPLE{X TRUE}}` is a relation literal (**Tutorial D** notation) denoting the relation whose only tuple is the 1-tuple whose attribute *X* has the value TRUE.
3. Joining the results of Steps 1 and 2 eliminates the tuples for which *c* evaluates to FALSE.
4. The final projection eliminates the attribute *X* added in Step 1, such that the result has the same heading as *r*.

26. 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 …

27. Aggregate Operators

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.

28. 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.

29. 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 …

30. To obtain C_ER from COURSE and EXAM_MARK:

```
EXTEND COURSE : { Exam_Result :=
                 ( EXAM_MARK JOIN
                    RELATION { TUPLE { CourseId CourseId } } )
                    { ALL BUT CourseId } }
{ CourseId, Exam_Result }
```

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

Explanation:

1. `RELATION { TUPLE { CourseId CourseId } }` is an open expression to be evaluated against each tuple *t* of `COURSE`. It denotes the relation whose only attribute is named `CourseId` and whose only tuple has as its only value that of the `CourseId` attribute of *t*. `CourseId CourseId` looks a bit funny, especially in black and white, so let me explain in more detail:

   In the Powerpoint 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 that tuple *t* of `COURSE`.
2. Joining that single-tuple relation with `EXAM_MARK` yields the relation consisting of just this tuples of `EXAM_MARK` that have the value of `CourseId` in tuple *t*. Note how this yields an empty relation for course C4. The result relation is known as the *image relation* of tuple *t* in `EXM_MARK`.
3. Projecting that on `{ ALL BUT CourseId }` eliminates the now redundant `CourseId` attribute—its value in each tuple is given by the `CourseId` attribute of `COURSE`.

"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.

31. 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** …

32. SUMMARIZE BY

This is the operator proposed by the ISBL team (and appearing in Baroque form in SQL). It 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 …

33. SUMMARIZE PER

This variety was proposed by Date and Darwen to address the problem just mentioned with SUMMARIZE BY. 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.

In practice, the highest and lowest members of types are not always defined and so MAX and MIN on empty relations are not always defined either.

34. 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 …

35. 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!

36. 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?

37. 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 …

38. 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.

39. Definition of NOT MATCHING

Tutorial D also has an operator, MATCHING, where *r* MATCHING *s* is equivalent to (*r* JOIN *s*){ *Hr* }, where *Hr* is a list of attribute names representing the heading of *r*. Thus, the result is the relation consisting of every tuple of *r* whose common attributes with *s* match those of at least one tuple of *s*. Thus, MATCHING is not a primitive operator even though its converse, NOT MATCHING, is.

MATCHING and NOT MATCHING both appeared (under different names) in ISBL, as generalisations of Codd's intersection and difference operators, respectively. The name *semijoin* was advanced for MATCHING in the late 1970s, presumably on the grounds that it does half the job, so to speak, of a join. The name *semidifference* came much later, as a companion to semijoin, but it's hard to think of this operator as doing half the job of difference.

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?

MINUS is **Tutorial D**'s difference operator. Like TIMES and INTERSECT it is not needed but is kept for historical reasons. It completes Codd's trio of counterparts of the basic set operators. You might like to use MINUS 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 or TIMES 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*. Similarly, *r1* INTERSECT r2 and *r1* TIMES *r2* are equivalent to *r1* JOIN *r2* for all the cases where they are defined.

Exercise: Define *r1* NOT MATCHING *r2* in terms of MINUS and other operators.

40. Constraints

Avoid the terrible mistake of thinking of a constraint as a condition applying to "data to be entered into the system".

You should be familiar with key constraints and foreign key constraints, for example. A key constraint on a relvar, such as KEY { StudentId } for IS_CALLED, specifies that no relation containing two or more tuples with identical values for the specified attribute(s) can ever be the value assigned to that relvar. Checking the constraint when tuples are inserted or updated entails looking at the current value of the relvar as well as the inserted or updated tuples.

A foreign key constraint, such as FOREIGN KEY { StudentId } REFERENCES IS_CALLED defined for relvar IS_ENROLLED_ON, requires that every value appearing for the foreign key in the referencing relvar appears also as a key value in the referenced relvar. It needs to be checked not only on inserts and updates to IS_CALLED in the example but also on deletions and updates to IS_ENROLLED_ON.

41. IS_EMPTY Example

Even here the notion of applying the constraint to "data being entered into the system" is misleading and inappropriate, even though it is obviously sufficient to check a tuple presented for insertion without looking at the current value of EXAM_MARK. Consider an update that adds 10 to every student's mark. That would violate the constraint if the existing value shows any student as having scored more than 90 in some exam.

By the way, **Tutorial D** does not provided special syntax for foreign key constraints. The one suggested in the notes for Slide 40 can be expressed simply as IS_EMPTY(IS_ENROLLED_ON NOT MATCHING IS_CALLED).

42. The End

One final remark: As I said, relational algebra, of itself, has nothing to do with databases and updating, but the examples clearly show its applicability to various problems in connection with databases and its use in a computer language addressing those problems: expressing queries and defining database constraints, for example. Note also that it can be used for updating too, by assignment of a relational expression to a relation variable, just as numerical expressions can be assigned to numerical variables. Indeed, the familiar update operators, INSERT, DELETE, UPDATE, are all defined in terms of such assignment. For example,

```
IS_CALLED := IS_CALLED WHERE NOT ( Name = 'Boris' ) ;
```
and
```
DELETE IS_CALLED WHERE Name = 'Boris' ;
```

are equivalent. As an exercise, you might like to express the following as direct assignments too:

```
INSERT IS_CALLED RELATION{TUPLE{StudentId SID('S6'), Name
'Eve'}} ;

UPDATE EXAM_MARK : { Mark := Mark + 5 } ;
```

End of Notes