The Perversion of the Relational Model

- A Tutorial in How to Implement a pre-1970's, pre-Hierarchic Record Filing System that appears to be Relational
- A Study to the art of Sabotage

(Original false title removed)

Hugh Darwen

This is an informal description of E.F. Codd's model [2] that was originally drafted as part my contribution to a special edition of the IEEE Annals of the History of Computing devoted to the history of relational model. It turned out to be too long and its level of detail was not thought to be suited to my article, which was to serve as the introductory article to that edition.

The terminology is that used in *The Third Manifesto* by Chris Date and myself. Our terminology is mostly the same as Codd's but we made a few changes in an attempt to clear up some matters that had given rise to confusion over the years. The notation used in my examples is taken from **Tutorial D** [3,4], a language Chris Date and I devised as an example for teaching purposes.

My description is in three main sections. First, in **Definitions,** I describe the objects that constitute a relational database. Then, in **Relational Algebra,** I describe the operators that are defined in the model for operating on those objects. Finally, **Database Integrity** covers the mechanism the model prescribes for maintaining consistency in a database.

Differences between the current definition and Codd's original are mentioned in footnotes.

Definitions

What's a relation?

Consider the sentence, "Student S1, named Anne, is enrolled on course C1". Because it is the kind of sentence of which we can say, "That's true", or "That's false", it denotes a *proposition*. A *predicate* is a generalized form of a proposition in which some *designators* (such as S1, Anne, and C1) are replaced by symbols denoting *parameters*,[1] as in "Student *StudentId*, named *Name,* is enrolled on course *CourseId.*" That sentence no longer denotes a proposition, but nevertheless it has the same declarative form as a proposition and thus denotes a predicate. Assuming the table in Figure 1 is to be interpreted according to this predicate, the first row of that table tells us that student S1 is named Anne and is enrolled on course C1, and the remaining rows similarly describe other enrolments.

[1] Also called free variables. We prefer "parameter" to avoid confusion with the term "variable" as used in computer languages (and in the relational model).

StudentId [*SID*]	Name [*NAME*]	CourseId [*CID*]
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 1: A tabular depiction of a relation for the predicate
"Student *StudentId*, named *Name,* is enrolled on Course *CourseId*."

Because the table in Figure 1 can be thus interpreted, it is in fact a tabular representation of a *relation* and each of its rows below the row of column headings represents a *tuple* that belongs to (i.e., is a member of) the *body* of that relation. The row of column headings represents the *heading* of that relation. Each column heading shows an *attribute* name sitting above a *type*[2] name, so we say that the relation has attributes StudentId of type SID, Name of type NAME, and CourseId of type CID. Because it has three attributes, we say it is a relation of *degree* three (equivalently a *ternary* relation) and because its body has five tuples we say its *cardinality* is five. Those tuples are also of degree three, so we call them 3-tuples when we want to mention that fact—the term *n*-tuple is used in general for a tuple with *n* components. To qualify as a tuple of a given relation, a tuple (a) must consist of exactly one *attribute value*, being a value of the indicated *type*, for each attribute of the heading, and (b) must result in a true proposition when its attribute values are substituted for their corresponding parameters of the predicate for that relation (in which case we say that the tuple *satisfies* the predicate). Moreover, a tuple that thus qualifies is, by definition, a member of that relation (in accordance with what logicians call the *Closed World Assumption*). So, if we are to believe Figure 1, we have to conclude that currently there are just five enrolments, involving four students and three courses.

A proposition derived from a predicate by substitution of each of the parameters is an *instantiation* of that predicate. The instantiations that are true constitute the *extension* of the predicate. Thus, we say that a relation represents the extension of a predicate in the way that I have described: each tuple provides the required attribute values for a true instantiation.

[2] Codd used the term "domain" here. There was initially some confusion surrounding his use of that term but it eventually became clear that the more familiar concept of type, as used in computer languages, was consistent with his requirements for domains.

Values, types, and operators

A *value* is a constant, unambiguously *designating* something. A *type* is a named set of values,[3] determining the *operators* that are defined in connection with its values. For example, the name INTEGER might refer to a type for whose values operators such as "+" are defined in the usual manner.

In Figure 1, the values depicted in the column headed StudentId are of type SID, the set of possible student identifiers. SID is a *scalar type,* which for present purposes is one that isn't a *tuple type* or a *relation type*. The tuples depicted by the rows below the heading in Figure 1 are values of the tuple type whose name (in **Tutorial D**) is TUPLE{StudentId SID, Name NAME, CourseId CID}. The braces indicate that what they enclose is a list denoting the elements of a set—specifically, a *heading*—and each element is an attribute, defined by its name and type. An alternative spelling for that type name is TUPLE{CourseId CID, StudentId SID, Name NAME}, because the order in which the attributes are listed carries no significance whatsoever. The entire relation depicted in Figure 1 is also a value, specifically a value of the relation type RELATION{StudentId SID, Name NAME, CourseId CID}. Note how the heading for the relation type is also the heading for the type of each of its tuples.

With a single exception, the relational model does not specify which scalar types are to be defined, but every type that is defined is available for use as the type of one or more attributes of a relation. The single exception is the scalar type, named BOOLEAN in **Tutorial D**, whose values are just the two truth values, TRUE and FALSE. It is required because the comparison operator, "=", is also required in connection with every type, such that the comparison $a = b$, where a and b denote values of the same type, yields TRUE if they denote the same value, otherwise FALSE. Equals comparison has to be defined for every type because the operators of the relational algebra depend on it and every type can be the type of an attribute.

In practice, of course, the DBMS is expected to provide a set of judiciously chosen *built-in types* along with facilities for users to define additional, *user-defined types*. The types SID, NAME, and CID are all user-defined types. The definition of type SID, for example, might specify that its values are represented by character strings (values of the built-in type CHAR), restricted to strings consisting of the capital letter S followed by from one to five decimal digits. Type CID is perhaps similarly defined, with upper-case C in place of S. Thus, we prevent "impossible" student identifiers and course identifiers from appearing anywhere in the database. We also protect users from accidentally giving queries that would involve comparing student identifiers with course identifiers—a comparison such as "StudentId = CourseId" would be illegal because the comparands are of different types.

---

[3] Note carefully that condition (a) precludes the possibility of either a pointer or a "null" appearing in place of an attribute value.

Relations in mathematics

Codd didn't invent relations. The term is used in mathematics for essentially the same concept, but Codd's treatment differs significantly from the usual mathematical treatment, and this difference represents one of Codd's great insights that made relations so applicable to the database problem.

Mathematicians don't use an expression like TUPLE{a 1, b 2} to denote a 2-tuple. Instead, they would write something like <1,2>, denoting an *ordered pair*. Thus <1,2> and <2,1> do not denote the same thing, whereas TUPLE{a 1, b 2} and TUPLE{b 2, a 1} do. Codd realised that if *n*-ary relations were to be used for database purposes, *n* could sometimes be quite large and then it would be unreasonable to expect users to remember the order of the attributes. As a result, he was careful not to define any ordering, either for the attributes of a heading, or for the tuples of a body. Thus, the table shown in Figure 2 represents exactly the same relation as that in Figure 1.

Name [*NAME*]	CourseId [*CID*]	StudentId [*SID*]
Cindy	C3	S3
Anne	C1	S1
Devinder	C1	S4
Boris	C1	S2
Anne	C2	S1

Figure 2: Another tabular depiction for the relation in Figure 1

Relation variables (relvars)

Suppose we use a name, ENROLMENT, for the relation in Figure 1, and the DBMS allows us to refer to it by that name. Suppose further that the relation assigned to that name can be replaced from time to time by a different relation of the same type. Then ENROLMENT is a relation variable,[4] or *relvar* for short. Under the *Principle of Uniformity of Representation,*[5] a relational database consists of relvars and nothing else. Updating the database, however it is done, results in at least one of those relvars having its value, a relation, replaced by a different

[4] Codd didn't use this term or its abbreviation. Instead he referred to "time-varying relations". Much of the ensuing literature used "relation" for both relations and relvars, causing some confusion. The same confusion sometimes arises when the term "table" does similar double duty in SQL literature.

[5] Codd called it *The Information Principle*

relation of the same type.

Relational Algebra

The relational algebra is a set of operators that are *closed over* relations, meaning that they allow further relations to be derived from given relations. Moreover, invocations of these operators do always yield relations, never anything else. The property of closure allows the DBMS's query language to support expressions of arbitrary complexity—an operand for an invocation of one of these operators can be a relation that is the result of some other invocation, just as, in the arithmetical expression (10+5)/3, the result of an invocation of "+" is the first operand to an invocation of "/".

Relational operators are based on the logical connectives and quantifiers of the predicate calculus. If a relational language supports relational operators corresponding to the connectives AND (conjunction), OR (disjunction), and NOT (negation), within certain well-defined limitations that I shall mention, and also existential quantification,[6] then it is *relationally complete*. The algebra whose operators I describe in the following subsections is just one of several that are equivalent in their expressive power and relationally complete.

The algebra whose operators I describe here is just one of several that are equivalent in their expressive power and relationally complete. Because it is relationally complete, it can be taken as a definition for relational completeness: a different algebra is relationally complete so long as it supports an equivalent expression for every invocation of every operator described here; a computer language is relationally complete if it does the same.

The notation used in my examples is taken from **Tutorial D** [3,4], a language Chris Date and I devised as an example for teaching purposes, but it must be emphasized that the choice of notation is not significant—it is the semantics of these operators that define the relational model, not the notation. Codd used mostly Greek letters and many textbooks have followed him in that respect. Of course the choice of notation is to some extent arbitrary, but **Tutorial D** is designed to be more in the style of typical programming languages that use English key words and, we hope, more digestible. It's also the one that I happen to be most familiar with, obviously. In fact, a relational database language doesn't have to use an algebraic notation at all. Codd himself proposed a notation in a style he called relational calculus, akin to first-order predicate calculus, using logical connectives and quantifiers and thereby raising the level of abstraction to one that is closer to what relations represent (extensions of predicates) by way of information.

The algebra described here incorporates certain improvements to Codd's relational algebra that the developers of ISBL [5] had found to be needed. ISBL (Information Systems

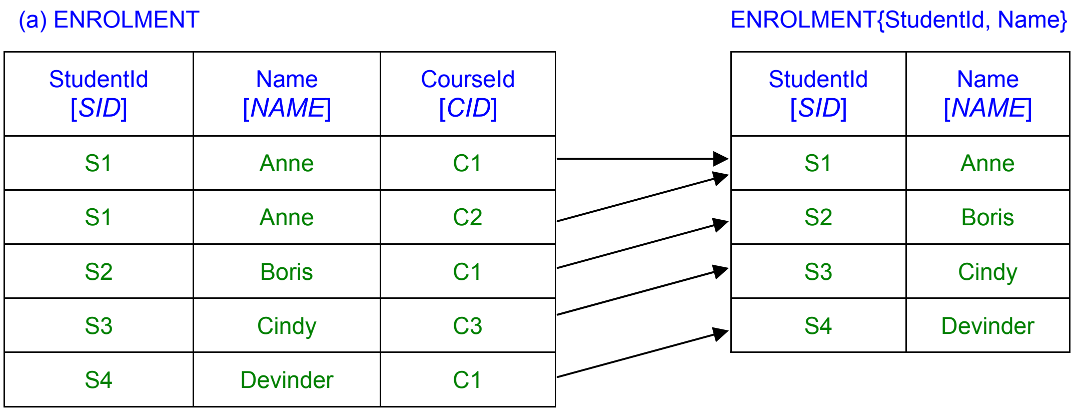
[6] For symmetry at least, one might expect to see universal quantification also mentioned here. Codd did include such an operator, which he called division, but his definition was questioned for being too restrictive and in any case, as he pointed out himself, it wasn't needed for completeness.

Base Language) was designed in the early 1970s by researchers at IBM UK's Scientific Centre in Peterlee, England, and implemented as PRTV (standing for "Peterlee Relational Test Vehicle", an unglamorous name imposed on the researchers by higher authorities in IBM for political reasons). Differences between the current definition and Codd's original are mentioned in end notes. *Rel* [6] is a freely available implementation of **Tutorial D**. Informal descriptions of the operators constituting this algebra now follow, using simple illustrative examples.

Projection (for existential quantification)

The predicate for ENROLMENT is "Student *StudentId* is called *Name* and is enrolled on course *CourseId*". The relation representing the extension of that predicate is depicted in Figure 1. By existential quantification we can derive the predicate "Student *StudentId* is called *Name* and is enrolled on some course".[7] The binary relation representing the extension of that predicate is obtained using projection. Figure 3, example (a), shows how: the expression ENROLMENT{StudentId, Name} denotes the relation derived from ENROLMENT by taking just the StudentId and Name values from ENROLMENT. The arrows show from which tuples in ENROLMENT the resulting tuples are derived. In the case of student S1, called Anne, two distinct tuples in ENROLMENT give rise to the same tuple in the result—the same tuple *never* appears more than once in the body of a relation.

Example (b) in the same figure shows a different projection of the same relation, this time using an alternative notation whereby we specify the attribute(s) to be excluded rather than those required.



StudentId [*SID*]	Name [*NAME*]	CourseId [*CID*]
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

StudentId [*SID*]	Name [*NAME*]
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder

Figure 3: Projection examples (continued on next page)

[7] A little more formally, "there exists a course *CourseId* such that student *StudentId* is called *Name* and is enrolled on *CourseId*". The abbreviated form, in which the symbol *CourseId* doesn't appear at all, emphasises that this parameter (free variable) has been bound by the quantifier and is thus no longer a parameter.

(b) ENROLMENT

StudentId [*SID*]	Name [*NAME*]	CourseId [*CID*]
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

ENROLMENT{ALL BUT Name}

StudentId [*SID*]	CourseId [*CID*]
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

Figure 3: Projection examples (continued from previous page)

Examples (a) and (b) illustrate an important point in database design. The information conveyed in the two results is equivalent to that conveyed in ENROLMENT, in the sense that every fact represented in one representation is also represented in the other; but in the two results of Figure 3 it is conveyed *without any repetition*: the name for student S1 is given just once instead of twice. Thus, if we assign the result of (a) to a new relvar, IS_CALLED, and that of (b) to another new relvar, IS_ENROLLED_ON, we can dispense with ENROLMENT. Then we will have eliminated a certain kind of redundancy from our database. Redundancy of that kind has been the subject of a great deal of study over the years, resulting in the definition of various *normal forms* for relvars. Design theory is a rich part of relational theory in general but it is built on top of the relational model, rather than being part of it, and is hence beyond the scope of this article.

Projection over no attributes is defined. For example, ENROLMENT{ } denotes a relation, of degree zero,[8] that is either empty or, as in the example at hand, contains just one tuple, the 0-tuple (i.e., the tuple having the empty set of components), signifying the existence of at least one course on which at least one student who has at least one name is enrolled.

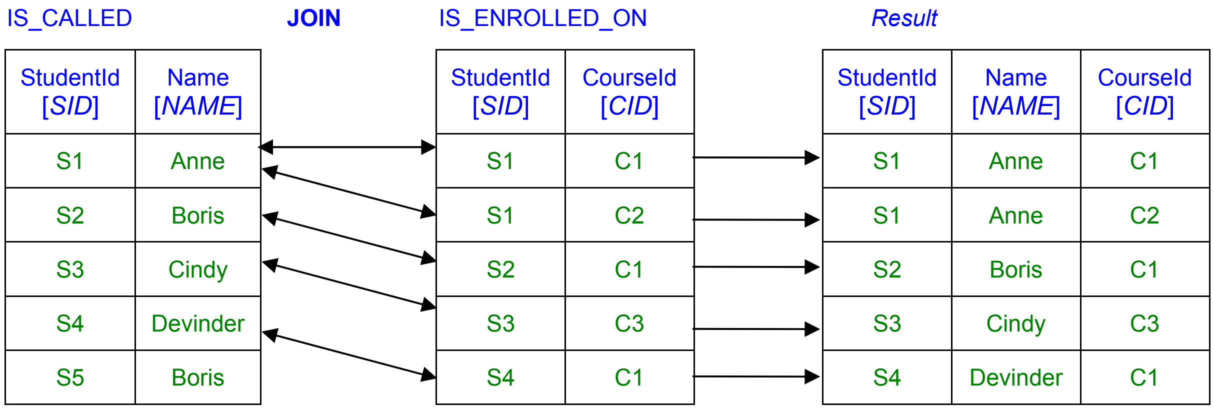
JOIN (for conjunction)

Let relvars IS_CALLED and IS_ENROLLED_ON be as just defined. Assume that IS_CALLED has been updated, as shown in Figure 4, by the addition of a tuple for student S5 who is not enrolled on any courses (so the predicate for IS_CALLED has been simplified just to "Student *StudentId* is called *Name*"). Using the dyadic operator JOIN we can derive the relation for the predicate "Student *StudentId* is called *Name* **and** Student *StudentId* is enrolled on course *CourseId*", this being the conjunction of the predicates for IS_CALLED and

[8] Codd did not recognize relations of degree zero. Their existence was noted by the ISBL developers.

IS_ENROLLED_ON. Notice that in normal parlance we would abbreviate that predicate to "Student *StudentId* is called *Name* and is enrolled on course *CourseId*", eliding the second mention of *StudentId* and confirming that the longer form just has two appearances of the same parameter. Thus, the corresponding relation has just three attributes, not four.

The result is shown in Figure 4. Note that it is the same as the original ENROLMENT relation (see Figure 1). The additional student S5 in IS_CALLED does not appear in this result because there is no tuple for S5 in IS_ENROLLED_ON.



IS_CALLED **JOIN** IS_ENROLLED_ON

StudentId [*SID*]	Name [*NAME*]
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

StudentId [*SID*]	CourseId [*CID*]
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

Result

StudentId [*SID*]	Name [*NAME*]	CourseId [*CID*]
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 4: Joining IS_CALLED with IS_ENROLLED_ON

JOIN, like the logical connective AND, is both commutative and associative. In the special case where the operands have no common attributes it is sometimes referred to as Cartesian product. However, this is not the usual mathematical operator of that name,[9] which is neither commutative nor associative. These nice properties of JOIN arise partly from Codd's great insight that I mentioned earlier, whereby the attributes of a relation are not ordered.

Antijoin[10] (limited support for negation)

Negation as such does not have a corresponding relational operator in this algebra. That's because, for example, the relation for "It is **not** the case that student *StudentId* is called *Name*" contains every tuple with heading {StudentId SID, Name NAME} that does not appear in

[9] The mathematical Cartesian product is defined for sets, taken pairwise, such that the Cartesian product of set *A* and *B*—*in that order*—denotes the set consisting of every *ordered* pair <*a*,*b*> such that *a* is a member of *A* and *b* is a member of *B*.

[10] The name, which is by no means universally accepted, wasn't used by Codd, who instead defined a *difference* operator, having the semantics of antijoin but requiring the operands to be of the same type. Using difference instead of antijoin, the example in Figure 5 becomes more complicated to express. Antijoin was proposed in ISBL, as "generalized difference".

IS_CALLED, far too many for computational purposes. However, the dyadic “antijoin” operator (Figure 5) admits negation when it is accompanied by conjunction and existential quantification, as in “Student *StudentId* is called *Name* **and** student *StudentId* is **not** enrolled on any course.” (Again, we can elide the second appearance of “student *StudentId*”, of course.)

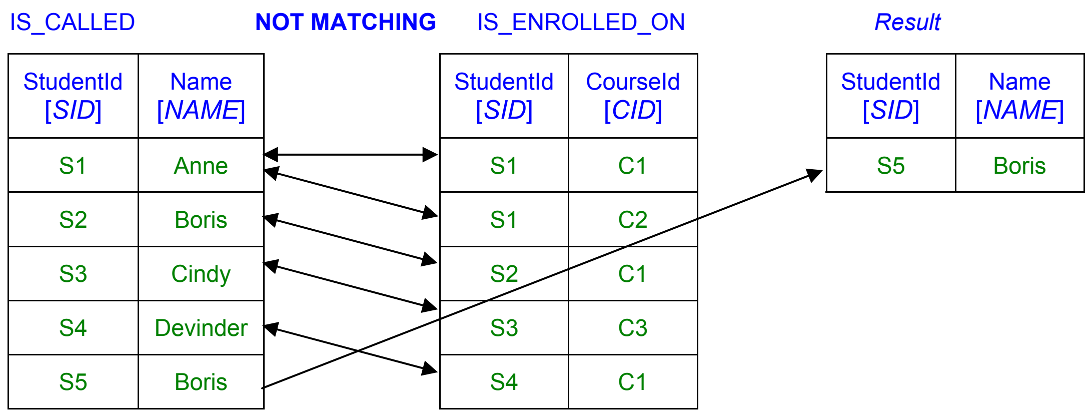


Figure 5: Antijoin of IS_CALLED with IS_ENROLLED_ON

As Figure 5 shows, **Tutorial D** calls this operator NOT MATCHING, appealing to the notion that the result consists of those tuples of the first operand that have no matching tuples in the second (where “matching” means comparing equal on each of the common attributes—just StudentId in the example).

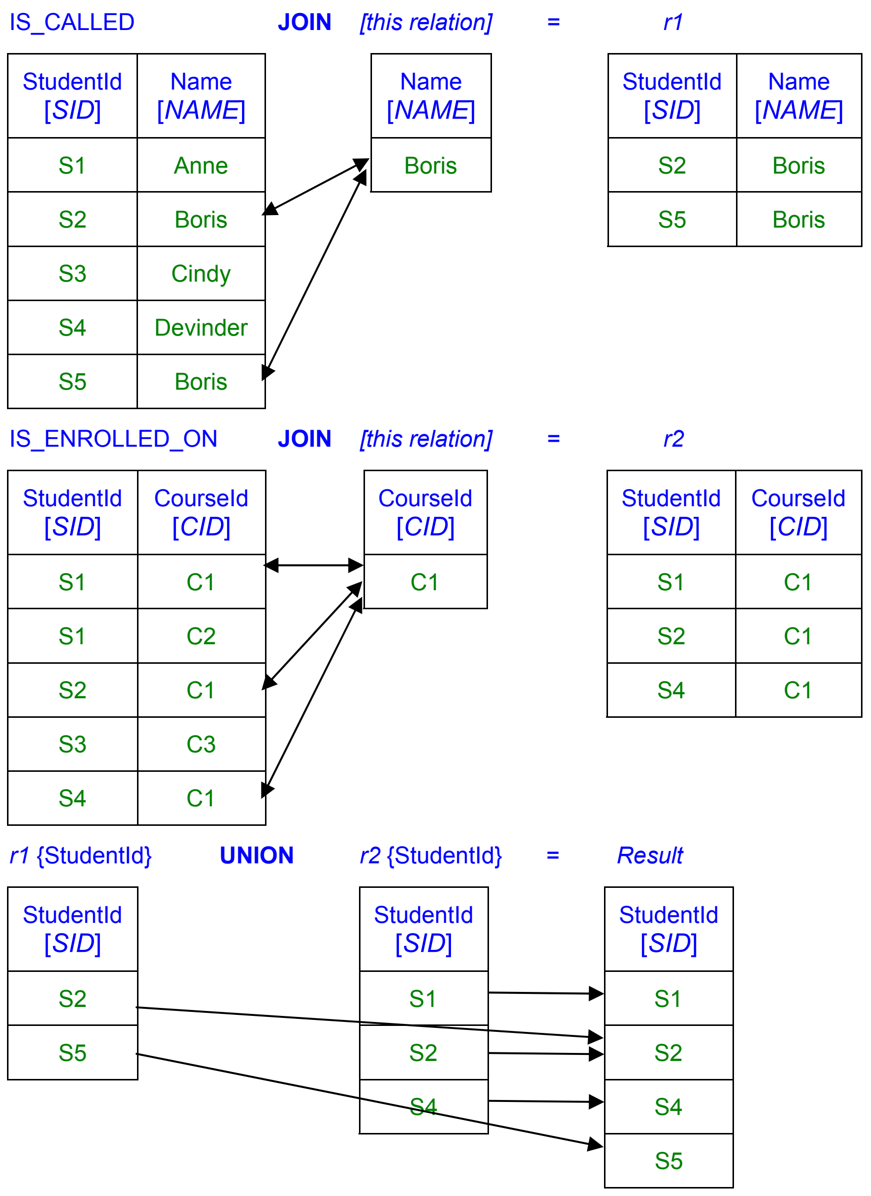
Unlike JOIN, antijoin is neither commutative nor associative, the result of an invocation being a relation whose body is a subset of the first operand.

UNION (limited support for disjunction)

Consider the predicate “**Either** student *StudentId* is called *Name* **or** student *StudentId* is enrolled on course *CourseId*”. The corresponding ternary relation is too large and not very useful. For example, for student S1 and course C1, it contains not only the tuple with name Anne but also tuples with all the other values of type NAME, because it is certainly true, for example, that S1 is either named Lancelot or is enrolled on C1. However, Codd realized that if two relations have the same heading, then their union, representing the disjunction of their predicates, would simply consist of each tuple that appears in either of them and can thus be easily computed.

Figure 6 shows the use of UNION, in combination with other operators, to obtain the unary relation for “Either student *StudentId* is named Boris or student *StudentId* is enrolled on course C1” (abbreviated in Figure 6). The two little relations headed *[this relation]* could be denoted by *relation literals*, for example RELATION {Name NAME} {TUPLE {Name NAME('Boris')}} and RELATION {CourseId CID} {TUPLE {CourseId CID('C1')}} in

Tutorial D, where the first pair of braces enclose a heading, the second a body.



IS_CALLED **JOIN** *[this relation]* = *r1*

StudentId [*SID*]	Name [*NAME*]
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

Name [*NAME*]
Boris

StudentId [*SID*]	Name [*NAME*]
S2	Boris
S5	Boris

IS_ENROLLED_ON **JOIN** *[this relation]* = *r2*

StudentId [*SID*]	CourseId [*CID*]
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

CourseId [*CID*]
C1

StudentId [*SID*]	CourseId [*CID*]
S1	C1
S2	C1
S4	C1

r1 {StudentId} **UNION** *r2* {StudentId} = *Result*

StudentId [*SID*]
S2
S5

StudentId [*SID*]
S1
S2
S4

StudentId [*SID*]
S1
S2
S4
S5

Figure 6: UNION of projections of joins for "Student *StudentId* is either named Boris or is enrolled on course C1"

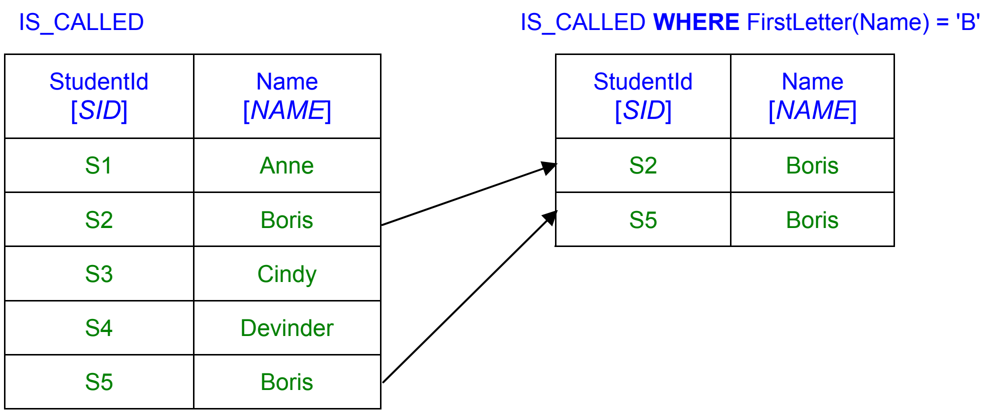
Like the logical connective OR, UNION is both commutative and associative.

Additional operators

Although the operators I have described thus far (projection, JOIN, antijoin, UNION) are theoretically sufficient for relational completeness, in practice two more (restriction and extension) are required for computational purposes and a further two (summarization and attribute renaming) are normally included for convenience.

The examples in Figures 7 and 8 assume that scalar operator FirstLetter is defined for values of type NAME, yielding the first letter of its operand.

Restriction (WHERE—see Figure 7) takes a relation, *r*, and returns the relation whose tuples are those of *r* that satisfy a specified condition.



IS_CALLED

StudentId [*SID*]	Name [*NAME*]
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

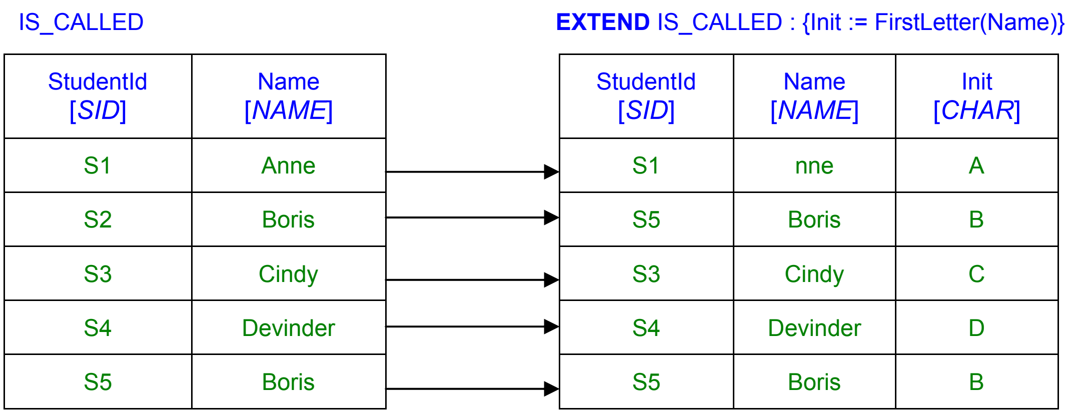
IS_CALLED **WHERE** FirstLetter(Name) = 'B'

StudentId [*SID*]	Name [*NAME*]
S2	Boris
S5	Boris

Figure 7: Restriction

In an *extension*[11] of *r* (Figure 8), each tuple is an extension of exactly one tuple *t* in *r*, consisting of the attribute values of *t* and a further attribute value derived from those of *t* by evaluation of a given expression.

[11] Codd's algebra omitted extension and attribute renaming. They were added in ISBL, which also introduced a limited form of summarization.



IS_CALLED

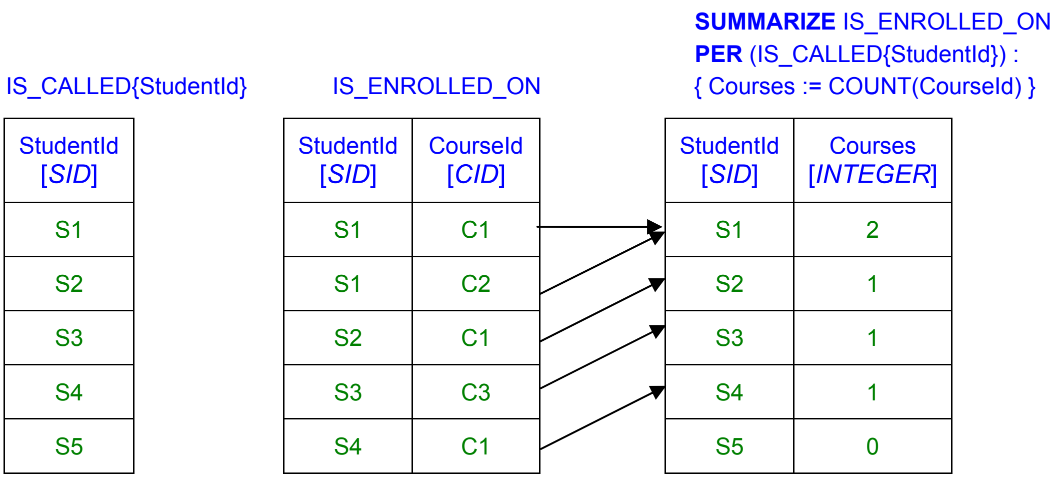
StudentId [*SID*]	Name [*NAME*]
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

EXTEND IS_CALLED : {Init := FirstLetter(Name)}

StudentId [*SID*]	Name [*NAME*]	Init [*CHAR*]
S1	nne	A
S5	Boris	B
S3	Cindy	C
S4	Devinder	D
S5	Boris	B

Figure 8: Extension

Summarization, which can be defined as a projection of a rather complicated extension, incorporates the use of aggregation to compute things like counts, sums, averages, maxima, and minima. The example in Figure 9 gives, for each student mentioned in IS_CALLED, the number of courses on which that student is enrolled.



IS_CALLED{StudentId}

StudentId [*SID*]
S1
S2
S3
S4
S5

IS_ENROLLED_ON

StudentId [*SID*]	CourseId [*CID*]
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

SUMMARIZE IS_ENROLLED_ON
PER (IS_CALLED{StudentId}) :
{ Courses := COUNT(CourseId) }

StudentId [*SID*]	Courses [*INTEGER*]
S1	2
S2	1
S3	1
S4	1
S5	0

Figure 9: Summarization

The self-explanatory operator, *attribute renaming* (Figure 10), can also be defined as a projection of an extension. Its usefulness is illustrated in Figure 11, where we obtain two renamings of the same relation, IS_CALLED, so that there is just one common attribute, StudentId, for the subsequent invocation of JOIN.

IS_CALLED

StudentId [*SID*]	Name [*NAME*]
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

IS_CALLED RENAME { Name AS Name1 }

StudentId [*SID*]	Name1 [*NAME*]
S1	Anne
S5	Boris
S3	Cindy
S4	Devinder
S5	Boris

Figure 10: Attribute renaming

Database Integrity

The relational model requires the DBMS to be largely responsible for a database's integrity—its consistency with the business rules of its owners. To this end, it further requires the DBMS to allow every such rule that the owners might wish to enforce to be expressed declaratively, as *integrity constraints*, in terms of permissible relvar values (as opposed to procedures governing invocations of update operators).

An integrity constraint defined for a database is a condition that the database must satisfy at all times, in order to keep it consistent with the possible real world situations that it might represent.

For example, assume the university wishes to record just one name for each of its students. Then suppose an attempt is made to record the name Eva for student S1, in addition to the existing name Ann, by adding the appropriate tuple to IS_CALLED. Figure 11 shows a relational expression whose result exposes the error. No tuple for student S2, for example, appears in this result because the only tuple for S2 in the result of the join has equal values for Name1 and Name2—loosely speaking, it is a tuple in IS_CALLED joined with itself, unlike those two tuples in Figure 11. The required constraint can be enforced by telling the DBMS that the expression in Figure 11 must always yield an empty relation.

```
( IS_CALLED RENAME { Name AS Name1 }
 JOIN
 IS_CALLED RENAME { Name AS Name2 } )
WHERE Name1 ≠ Name2
```

StudentId [*SID*]	Name1 [*NAME*]	Name2 [*NAME*]
S1	Anne	Eva
S1	Eva	Anne

Figure 11: Discovering a student with two names
(note the use of RENAME to make just StudentId the common attribute for JOIN)

Tutorial D accordingly provides an operator IS_EMPTY(*r*), where *r* is an arbitrary relational expression, and allows such expressions to appear in constraint declarations.

In practice, for the most commonly required forms of constraint, we expect more convenient ways—"shorthands"—for declaring them. In the example at hand, the constraint effectively says that IS_CALLED must never contain more than one tuple with the same StudentId value, in which case {StudentId} is said to be a key for that relvar, expressed in **Tutorial D** by including the specification KEY {StudentId} in that relvar's definition. We use braces here because a key, which in general can consist of several attributes, is a subset of the heading of the relvar to which it applies (and is therefore a set). For example, {StudentId, CourseId} is a key[12] for ENROLMENT—it is also a key for IS_ENROLLED_ON, but the specification is in a sense redundant because by definition the same tuple cannot appear more than once in a relation.

Another common kind of constraint is the *inclusion dependency* (see reference [1]). For example, the university probably requires enrolments to be restricted to students that are registered with it, which might translate to a requirement that every StudentId value currently appearing in IS_ENROLLED_ON must also currently appear in IS_CALLED. It's called an inclusion dependency because it can be expressed using the set comparison operator "⊆" ("is a subset of", or "is included in"). The requirement just mentioned could then be expressed as IS_ENROLLED_ON{StudentId} ⊆ IS_CALLED{StudentId}. Alternatively, it could be formulated as IS_EMPTY(IS_ENROLLED_ON NOT MATCHING IS_CALLED).

When the common attributes of an inclusion dependency constitute a key for the "including" relvar (IS_CALLED in the example), then the constraint is also variously called a

[12] As well as having the uniqueness property described here, a key is required, by definition, to be *irreducible*, meaning that no proper subset of a key has that uniqueness property. The term *superkey* is used for any subset of a relvar's heading that is a superset of a key. As the constraint implied by a KEY specification cannot enforce irreducibility, it is better referred to as a superkey constraint.

referential constraint or a *foreign key constraint*, the relvars involved in it are called the *referencing* relvar (IS_ENROLLED_ON in the example) and the *referenced* relvar (IS_CALLED), and the common attributes constitute a *foreign key* for the referencing relvar.

AND THAT'S ALL YOU NEED …

… apart, of course, from all the model-independent features that every database needs, such as provisions for database creation and destruction, recovery, security and authorization, transaction support, and so on. A database language conforms to the relational model if it somehow supports all the features I have described and importantly, under the aforementioned *Principle of Uniformity of Representation* does *not* permit the presence in the database of variables other than relation variables, as these are not necessary and would just lead to the kind of needless complexity that the model was designed to avoid.

REFERENCES

[1] Marco A. Casanova, Ronald Fagin, Christos A. Papadimitriou: "Inclusion dependencies and their interaction with functional dependencies", PODS '82, Proceedings of the first ACM SIGACT-SIGMOD symposium on Principles of Database Systems, pages 171-176. ACM, New York, NY, USA.

[2] E.F. Codd: "A Relational Model of Data for Large Shared Data Banks", *CACM 13,* No. 6 (June 1970). Earlier, Codd had published a preliminary version: "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks", IBM Research Report RJ599 (August 9th, 1969).

[3] Hugh Darwen and C.J. Date: *The Third Manifesto* (PDF available at www.thethirdmanifesto.com, along with a language definition for **Tutorial D**)

[4] C.J. Date and Hugh Darwen: *Databases, Types, and The Relational Model: The Third Manifesto.* Reading, Mass.: Addison-Wesley (3rd edition, 2007)

[5] P.A.V. Hall, P. Hitchcock,and S.J.P. Todd: "An Algebra of Relations for Machine Computation," Conf. Record of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, Calif. (January 1975)

[6] Dave Voorhis: *Rel: An Implementation of Date and Darwen's* ***Tutorial D*** *Database Language,* available at dbappbuilder.sourceforge.net/Rel.php)