3 Predicates and Propositions

3.1 Introduction

In Chapter 1 of the theory book I defined a database to be "…an *organised*, machine-readable collection of *symbols*, to be *interpreted* as a *true* account of some *enterprise*." I also gave this example (extracted from Figure 1.1):

StudentId	Name	CourseId
S1	Anne	C1

I suggested that those green symbols, organised as they are with respect to the blue ones, might be understood to mean:

"Student S1, named Anne, is enrolled on course C1."

That is how Chapter 3 of the theory book started. It continued with the following paragraph:

> In this chapter I explain exactly how such an interpretation can be justified. In fact, I describe the general method under which data organized in the form of relations is to be interpreted—to yield *information*, as some people say. This method of interpretation is firmly based in the science of *logic*. Relational database theory is based very directly on logic. Predicates and propositions are the fundamental concepts that logic deals with.

The remainder of the chapter could be said to apply equally well to the interpretation of SQL databases were it not for SQL's use of a logic based on three truth values instead of the usual two, this arising from its special construct referred to as `NULL`. Many of the effects of this intrusion have already been examined in Chapter 2. Here I make further observations that arise in connection with the corresponding sections of this chapter in the theory book.

3.2 What Is a Predicate?

Consider the declarative sentence—a proposition—that is used to introduce this topic in the theory book:

"Student S1, named Anne, is enrolled on course C1."

Recall that the terms S1, Anne, and C1 are *designators*, each referring unambiguously to a particular thing. The chapter later explains how a tuple can provide values—*attribute* values—to be interpreted as designators to be substituted for the corresponding parameters of a predicate. Thus, this sentence might be represented by the tuple denoted in **Tutorial D** by `TUPLE{StudentId SID('S1'), Name NAME('Anne'), CourseId CID('C1') }`. As SQL allows `NULL` to appear wherever a value can appear, we have to entertain the notion that the row denoted in SQL by `( SID('C1'), NULL, CID('C1') )` might represent some sentence. (*Aside:* SQL does not use attribute names to connect values to their corresponding parameters. Instead, the correspondence is determined by position and I have assumed that the parameters are to be considered in the order *StudentId, Name, CourseId*. *End of aside.*) Now, could that sentence be "Student S1, named `NULL`, is enrolled on course C1"? Well, no, because `NULL` is not a name and really doesn't designate anything. The row might instead represent the sentence "Student S1, whose name is not known, is enrolled on course C1". But that sentence contains nothing that can be regarded as a designator substituted for the parameter *Name*.

If we now recast this into two simpler sentences, as in the theory book, we will get something like, "Student S1's name is not known" and "Student S1 is enrolled on course C1". Let's now try replacing S1 by `NULL` in the second of those:

Example 3.1:

"Student `NULL` is enrolled on course C1."

Again that doesn't make sense. If `NULL` is *always* to be interpreted as meaning "some value should appear here but we don't know which", then perhaps the sentence should be "Some student, whose student identifier is not known, is enrolled on course C1." But again, that sentence contains nothing that can be regarded as a designator substituted for the parameter *StudentId*.

Similarly, `NULL`, might appear in place of C1:

Example 3.2:

"Student S1 is enrolled on course `NULL`."

and again we have to reject that and write instead, perhaps, “Student S1 is enrolled on some course, whose course identifier is not known”, or perhaps, “Student S1 is enrolled on some course but we don’t know which one”. The row `(SID('S1'), NULL)` could indeed mean either of those and either would be consistent with the notion of “some value appears here but we don’t know which”. However, in certain operations, such as the “outer joins” that we shall meet in Chapter4, SQL uses that very row to mean “Student S1 is not enrolled on any courses”, which, although perhaps a more likely interpretation in practice, is *not* consistent with the meaning “some value appears here but we don’t know which” (nor with the fact that SQL would not regard two such students as being enrolled on the same set of courses).

3.3 Substitution and Instantiation

Section 3.2 shows how `NULL` might appear in substitution for a parameter of a predicate and how it might thus participate in instantiation of that predicate to yield a proposition. Now consider instantiations of the dyadic predicate $a < b$. As well as instantiations such as $5 < 10$ (a *true* one) and $9 < 6$ (a *false* one), we now have to entertain the possibility of instantiations such as 5 < `NULL`, `NULL` < 6, and `NULL` < `NULL`. In SQL these comparisons evaluate to that intrusive truth value, *unknown*. Now, Section 3.2 in the theory book goes on to explain that the *extension* of a predicate consists exactly of those instantiations of it that evaluate to *true*, from which we can conclude, of every instantiation that does not appear in the extension, that it is *false*, in which case it must appear instead in the extension of the negation of that predicate. In SQL, then, the instantiation 5 < `NULL`, for example, cannot be considered to appear in either the extension of $a < b$ or NOT ($a < b$). Or so it would appear.

3.4 How a Table Represents an Extension…

…or does it? The theory book here describes how each tuple in a relation represents a true instantiation of some predicate and each true instantiation is represented by some tuple in that relation. Thus, a relation represents an extension, its body containing exactly one tuple corresponding to each element of the extension.

It is true that some SQL tables can be interpreted in this way but it is also true that some SQL tables cannot. In fact there are at least two distinct ways in which an SQL table cannot be thus interpreted:

a) In SQL it is possible for the same row to appear more than once in a table. Moreover, if tables *t1* and *t2* differ only in the number of appearances of some row, then that difference is significant—they are not the same table.

b) Although I have noted that in SQL the instantiation 5 < `NULL` cannot be considered to appear in either the extension of *a* < *b* or NOT (*a* < *b*), the row (5, `NULL`) *can* appear in a table. What could be the corresponding predicate? It would have to be some dyadic predicate, *P*(*a*, *b*) say, such that *P*(5, `NULL`) is *true*. But if `NULL` stands for "some value but we don't know which", how could that row appear in the same table as, say, (6, 12)? If (6, 12) means "6 is related to 12" then (5, `NULL`), in relational theory, would have to mean that 5 is related to `NULL` in that same way. But it can't, because `NULL` doesn't designate anything. If instead it means "5 is related to something whose identity is unknown", then we have a sentence in which nothing appears in substitution for the parameter *b*.

3.5 Deriving Predicates from Predicates

The corresponding section in the theory book describes how predicates can be derived from predicates using (a) the logical connectives of the propositional calculus, such as AND, OR, and NOT, and (b) quantifiers, such as "there exists" (∃) and "for all" (∀). Here I examine how SQL's truth value, *unknown*, intrudes on those connectives and quantifiers.

Logical Connectives

For these I give SQL's extended truth tables in which the symbol ∪, for *unknown*, appears along with the usual T and F.

Negation (*`NOT`*, ¬)

p	¬*p*
T	F
U	U
F	T

Figure 3.1: The SQL Truth Table for Negation

We now have three rows instead of just two. As you can see, ¬*p* is defined as in two-valued logic (2VL) when *p* is either *true* or *false*, but ¬(*unknown*) is *unknown*.

Other monadics

In 2VL there are just 4 (2^2) monadic operators, of which negation is really the only "useful" one. When a third truth value is introduced we have 27 (3^3) monadics and SQL gives names to several of these in addition to `NOT` for its version of negation. Some of these are shown in Figure 3.1a.

p	*p* **IS TRUE**	*p* **IS UNKNOWN**	*p* **IS FALSE**	*p* **= TRUE**	*p* **= UNKNOWN**	*p* **= FALSE**
T	T	F	F	T	U	F
U	F	T	F	U	U	U
F	F	F	T	F	U	T

Figure 3.1a: Truth Tables for Some Other SQL Monadics

Note that under none of the "`IS`" operators shown in Figure 3.1a does a truth value in the first column map to *unknown*—contrast this with the treatment of "`=`". In addition to those three SQL also has *p* `IS NOT TRUE`, *p* `IS NOT UNKNOWN`, and *p* `IS NOT FALSE`, equivalent to `NOT(`*p* `IS TRUE)`, `NOT(`*p* `IS UNKNOWN)`, and `NOT(`*p* `IS FALSE)`, respectively. The test *x* `=` *x* `IS UNKNOWN` can be useful in cases where it evaluates to `TRUE` when *x* `IS NULL` does not—for examples, see the **Effects of `NULL`** in Chapter 2, Section **2.10, Types and Representations**.

We turn now to the dyadic operators, noting that with three truth values there are now 19,683 (3 to the power 3^2) all told, compared with just 16 (2 to the power 2^2) in 2VL. SQL directly supports (i.e., has names for) just eight of these, including counterparts of conjunction, disjunction, and—surprisingly—implication (which, as we shall see, appears to have been included in the language by accident).

Conjunction (AND, ∧)

p	*q*	*p* ∧ *q*
T	T	T
T	U	U
T	F	F
U	T	U
U	U	U
U	F	F
F	T	F
F	U	F
F	F	F

Figure 3.2: The SQL Truth Table for AND

Now we have nine rows (3^2) instead of just four (2^2). Again, when *unknown* is not involved, the rows are as for 2VL. Also, when anything is paired with *false*, the result is *false*, as in 2VL. Our intuition, that "*p* and *q*" is true exactly when both operands are true, is preserved.

Disjunction (`OR`, ∨)

p	q	$p \lor q$
T	T	T
T	U	T
T	F	T
U	T	T
U	U	U
U	F	U
F	T	T
F	U	U
F	F	F

Figure 3.3: The SQL Truth Table for Disjunction

Again we have nine rows instead of just four and again, when *unknown* is not involved, the rows are as for 2VL. Also, when anything is paired with *true*, the result is *true*, as in 2VL. Our intuition, that "*p* or *q*" is true exactly when at least one operand is true, is preserved.

Now, in the theory book it is noted that disjunction could equally well be defined in terms of conjunction and negation, as

$$p \lor q \equiv \neg(\neg p \land \neg q)$$

and the truth table in Figure 3.4 of that book is given as proof of that equivalence. The question arises, does the same equivalence hold in SQL? To answer that we need to look at the revised Figure 3.4.

p	q	$\neg p$	$\neg q$	$\neg p \land \neg q$	$\neg(\neg p \land \neg q)$
T	T	F	F	F	T
T	U	F	U	F	T
T	F	F	T	F	T
U	T	U	F	F	T
U	U	U	U	U	U
U	F	U	T	U	U
F	T	T	F	F	T
F	U	T	U	U	U
F	F	T	T	T	F

Figure 3.4: SQL Disjunction in Terms of SQL Negation and SQL Conjunction

As you can see, the final column is the same as in Figure 3.3, so that equivalence does also hold in SQL.

Conditionals

At first sight SQL does not appear to have a single operator for expressing logical implication. In this respect it would be in common with most programming languages, including **Tutorial D**. However, standard SQL defines a partial ordering for its three truth values, under which *false* is deemed to precede *true*. Thus, the comparisons $p < q$, $p > q$, $p <= q$, and $p >= q$ are all supported in standard SQL (in addition to $p = q$, of course).

Now, in Section 3.5 of the theory book it is noted that in 2VL $p \rightarrow q$ is equivalent to $\neg p \vee q$. Study of Figure 3.5 reveals that $\neg p \vee q$ does indeed equate to $p \rightarrow q$ when neither operand is *unknown*, and the same is true of p <= q! (It is the pronunciation, "is less than or equal to", rather than "implies", that led to my observation that SQL appears to include direct support for a 3VL form of implication *by accident*.)

p	*q*	¬*p*	¬*p* ∨ *q*	*p* <= *q*
T	T	F	T	T
T	U	F	U	U
T	F	F	F	F
U	T	U	T	U
U	U	U	U	U
U	F	U	U	U
F	T	T	T	T
U	U	U	U	U
F	F	T	T	T

Figure 3.5: The SQL Truth Tables for ¬*p*∨ *q* and *p* <= *q*

Note, however, that p <= q is not equivalent to $\neg p \vee q$. Intuitively, we understand that "*p* implies *q*" is true whenever q is true. This holds for $\neg p \vee q$ but not for p <= q, as the row for p = U and q = T shows. The U in the last column for that row arises from SQL's general rule that whenever an operand of a comparison is NULL, the result is *unknown*—and NULL, when it is the result of evaluating a Boolean expression, is considered synonymous with *unknown*. In fact, Figure 3.5 gives a demonstration of the fact that SQL is not always faithful to its own concept, that NULL represents "a value exists here but we don't know which value". What U really means when it appears in the column for p <= q is that <= is *undefined* for that particular pair of truth values.

The biconditional $p \leftrightarrow q$ can be expressed in **Tutorial D** by $p = q$ and the same is true of SQL. The question then arises as to whether, in SQL, $p = q$ is equivalent to $(\neg p \lor q) \land (\neg q \lor p)$. This matter is investigated in the truth table of Figure 3.6.

p	q	$\neg p \lor q$	$\neg q \lor p$	$(\neg p \lor q) \land (\neg q \lor p)$	$p = q$
T	T	T	T	T	T
T	U	U	T	U	U
T	F	F	T	F	F
U	T	T	U	U	U
U	U	U	U	U	U
U	F	U	T	U	U
F	T	T	F	F	F
F	U	T	U	U	U
F	F	T	T	T	T

Figure 3.6: SQL $p = q \equiv (\neg p \lor q) \land (\neg q \lor p)$

As you can see, the equivalence does hold in SQL, but only because SQL treats *unknown* as not equal to—i.e., not the same truth value as—itself! This treatment of $p = q$ is consistent with the general rule that applies when `NULL` is an operand of a comparison in SQL.

Figure 3.6a similarly investigates whether $p = q$ is equivalent to $(p <= q) \land (q <= p)$, and as you can see, the equivalence again holds in SQL.

p	q	$p <= q$	$q <= p$	$(p <= q) \land (q <= p)$	$p = q$
T	T	T	T	T	T
T	U	U	U	U	U
T	F	F	T	F	F
U	T	U	U	U	U
U	U	U	U	U	U
U	F	U	U	U	U
F	T	T	F	F	F
F	U	U	U	U	U
F	F	T	T	T	T

Figure 3.6a: SQL $p = q \equiv (p <= q) \land (q <= p)$

Quantification

To quantify something, as the theory book has it, is to state its quantity, to say how many of it there are. For example, in **Tutorial D** the expression `COUNT(r)` denotes the number of tuples in the relation *r*, to be interpreted as the number of objects represented by those tuples that satisfy a predicate that *r* is considered to represent. Universal quantification—stating that something is true of all objects under consideration—is involved in expressions such as

- `AND(r,c)`, meaning that all objects that satisfy a predicate for *r* also satisfy the condition (another predicate) *c*, and
- `IS_EMPTY(r)`, meaning that no object satisfies a predicate for *r*—in other words, every object satisfies the negation of that predicate.

Existential quantification—stating that something is true of at least one object under consideration—can be expressed by `OR(r,c)`, meaning that at least one object that satisfies a predicate for *r* also satisfies *c*, and `IS_NOT_EMPTY(r)`.

The names for the aggregate operators AND and OR reflect the facts that when we confine our attention to finite sets, universal and existential quantification are equivalent to repeated invocations of dyadic AND and dyadic OR, respectively. Note that `AND(r,c)` is equivalent to `COUNT(r) = COUNT(r WHERE c)`, and `OR(r,c)` is equivalent to `COUNT(r WHERE c) > 0` and also to `IS_NOT_EMPTY(r WHERE c)`.

Quantification also appears in various guises in SQL, but its meaning is muddied by those same two violations of relational theory that we have already seen muddying the waters: duplicate rows and `NULL`. For example, SQL's `(SELECT COUNT(*) FROM` *r*`)`, a so-called scalar subquery (because it is an expression denoting a table with one row and one column, enclosed in parentheses), denotes the number of rows in the table *r*, but can we really say that this represents the number of objects that satisfy a predicate for *r*, if the same row can be counted more than once, or if `NULL` appears in place of a column value in some row of *r*? In fact, what might it mean to say that a row does or does not *satisfy* a predicate? In 2VL we say that object *a* satisfies predicate *P*(*x*) exactly when *P*(*a*) is *true*. Does this still hold in 3VL, or might SQL deem *a* to satisfy *P*(*x*) also when *P*(*a*) is *unknown*? Well, it turns out that SQL uses both interpretations, depending on the context, as we shall discover.

*SQL counterparts of **Tutorial D** quantifications*

Consider **Tutorial D**'s `AND(`*r*`,`*c*`)`, where *r* is a relation and *c* is a condition that is applicable to tuples of *r*. This variety of `AND` is an aggregate operator in **Tutorial D**. The expression evaluates to *true* when every tuple in *r* satisfies *c*, otherwise *false*. Looking for an SQL counterpart of this expression, we are faced with a plethora of possibilities. It is a salutary exercise to examine the apparent choices to see which, if any, is the best fit. For aggregation SQL provides what it calls "aggregate functions". These are counterparts, not to **Tutorial D**'s aggregate operators but rather to its constructs such as `SUM(x)` that can appear in invocations of `SUMMARIZE`. Aggregate functions are used in several of the candidates we shall examine. It is important to bear in mind two general rules that apply to aggregation in SQL. The first is that appearances of `NULL` are always excluded, such that, for example `SUM(x)` evaluates to 3 when summing 1, 2 and `NULL`, even though `1+2+CAST(NULL AS INTEGER)` evaluates to `NULL`. The second is that, except in the case of `COUNT`, aggregation over the empty set always yields `NULL`, even though the sum of no integers, for example, really should be zero and the `AND` of no truth values should be `TRUE`. With that in mind, let us now look at some of the possibilities for determining whether condition *c* is satisfied by every row of table *r*.

1. `(SELECT EVERY(`*c*`) FROM` *r*`)`
   The result is *false* if *c* evaluates to *false* for at least one row of *r, unknown* if *c* evaluates to *unknown* for each row of *r* (including the case where *r* is empty), otherwise *true*. Note that the treatment thus differs from `AND(`*r*`,`*c*`)` when *r* is empty, the result being *unknown* instead of *true*. Note also that the result can be *true* even when *c* does not evaluate to *true* for every row of *r*, namely, when *c* is *true* for at least one row and *unknown* for each of the others. Here, then, we can observe that a row appears to satisfy *c* if *c* evaluates to either *true* or *unknown*.

2. `(SELECT MIN(`*c*`) FROM` *r*`)`
   This is available as a result of the partial ordering of truth values previously mentioned. It is equivalent to (`SELECT EVERY(`*c*`) FROM r`).

3. `(SELECT EVERY(c IS TRUE) FROM r)`
   Here we are explicitly stating that we deem *r* to satisfy *c* only when *c* is *true* for *r*. But it is still the case that the expression yields *unknown* when *r* is empty.

4. `TRUE =ALL (SELECT c FROM r)`
   Here we are using the unusual construct SQL calls a "quantified comparison predicate". For example, the comparison `X =ALL (SELECT Y FROM T)` results in *true* if `X = Y` is *true* for every row of *r* (including the case where *r* is empty), *false* if `X = Y` is *false* for at least one row of *r*, otherwise *unknown*. (Note that the word `ALL` is attached to the comparison operator, not the table expression that follows it. `ALL (SELECT Y FROM T)` is not a legal expression in SQL.) So `TRUE =ALL (SELECT c FROM r)` yields the same result as `AND(r,c)` exactly when *c* is *true* for every row of table *r*, but otherwise it can yield either *unknown* or *false*.

5. `TRUE =ALL (SELECT c IS TRUE FROM r)`
   At last we have an expression that yields the same result as `AND(r,c)` when *c* is *true* for every row of table *r* and otherwise yields *false*.

We can conduct a similar investigation in connection with **Tutorial D**'s `OR(r,c)`:

1. `(SELECT SOME(c) FROM r)`
   The result is *true* if *c* evaluates to *true* for at least one row of *r*, *unknown* if *c* evaluates to *unknown* for each row of *r* (including the case where *r* is empty), otherwise *false*. The treatment differs from `OR(r,c)` in the case where *r* is empty and the result is *unknown*—you might find that a bit strange when you consider that if *r* contains no rows, then obviously there doesn't exist a row in *r* that satisfies *c*.

2. `(SELECT MAX(c) FROM r)` and `(SELECT ANY(c) FROM r)`
   These are both equivalent to (`SELECT SOME(c) FROM r`). `ANY` is just an alternative spelling for `SOME`.

3. `(SELECT SOME(c IS TRUE) FROM r)`
   This is also equivalent to (`SELECT SOME(c) FROM r`). The addition of `IS TRUE` has no effect this time.

4. `TRUE =SOME (SELECT c FROM r)`
   This differs from (`SELECT SOME(c) FROM r`) because it yields *false* instead of *unknown* when *r* is empty and also because it yields *unknown* when *c* evaluates to *false* for at least one row of *r* and *unknown* for all the others.

5. `TRUE =SOME (SELECT` *`c`* `IS TRUE FROM` *`r`*`)`
   Similar to our fifth candidate for `AND(`*`r`*`,`*`c`*`)`, we finally have an expression that yields the same result as `OR(`*`r`*`,`*`c`*`)` when *c* is *true* for some row of table *r* and otherwise yields *false*.

For **Tutorial D**'s `IS_EMPTY(`*`r`*`)` SQL has `NOT EXISTS(`*`r`*`)`, which yields *true* whenever *r* is empty, otherwise *false*. Note, then, that `NOT EXISTS(`*`r`* `WHERE FALSE)` is not equivalent to `(SELECT EVERY(FALSE) FROM` *`r`*`)`, because of the difference in treatment of the empty table.

Similarly, for `IS_NOT_EMPTY(`*`r`*`)` SQL has `EXISTS(`*`r`*`)`, which yields *False* whenever *r* is empty, otherwise *True*. Note, then, that `EXISTS(`*`r`* `WHERE TRUE)` is not equivalent to `(SELECT SOME(TRUE) FROM` *`r`*`)`. Note also that `EXISTS(`*`r`* `WHERE` *`c`*`)` evaluates to *false* in the case where *r* is not empty, *c* evaluates to *unknown* for at least one row of *r*, and *c* evaluates to *false* for every other row. Thus, although SQL uses the name `EXISTS` for this operator, it is not the 3VL existential quantifier. Similarly, `NOT EXISTS(`*`r`* `WHERE NOT (`*`c`*`))` does not in general express universal quantification in 3VL.

Historical Notes

It is commonly believed that the term Structured Query Language, sometimes taken to be the full name for SQL, is inspired by the SELECT-FROM-WHERE structure. This may be the case, but it is not clear whether that was the intention of the authors of SEQUEL. The Abstract for that paper gives a clue: "Moreover, the SEQUEL user is able to compose these basic templates [SELECT-FROM-WHERE templates] in a structured manner to form more complex queries." That "structured manner" might have referred to SEQUEL's support for nesting one SELECT-FROM-WHERE structure within another.

The syntax SELECT * FROM was not included in SEQUEL because the SELECT clause itself was optional, as was the key word FROM. Thus, SQL expressions such as SELECT * FROM T1 and SELECT * FROM T1, T2 could be written as just T1 and T1, T2 in SEQUEL. The shorthand TABLE *t* was added to the SQL standard in 1992 but remains an optional conformance feature.

The monadic operators IS TRUE, IS FALSE, IS UNKOWN and their negated counterparts IS NOT TRUE, IS NOT FALSE, IS NOT UNKNOWN were added to the language in SQL:1992. They remain optional conformance features.

Support for comparison operators on values of type BOOLEAN, along with the aggregate functions EVERY, SOME, ANY, and MAX and MIN on Booleans, arrived in SQL in the 1999 edition of the international standard, as already noted in Chapter 2. The partial ordering of truth values was perhaps partly a consequence of SQL's treatment of NULL when the rows of a table are to be placed in some specified order (typically by use of an ORDER BY clause). For example, suppose that rows of ENROLMENT are to be placed in alphabetical order of NAME, for which NULL appears in some row. Does this row appear before the rows for Anne or after the rows for Zack? When the first edition of the SQL standard (1986) was being drafted it was discovered that existing implementations were divided fairly evenly between those that placed NULL first in the ordering and those that placed it last. Rather than toss a coin to decide which implementations would be deemed in conformance, the committee decided not to legislate on this matter. When BOOLEAN was added in 1999, the treatment of comparisons on values of this type was at least consistent with that decision. It was also consistent with the existing treatment of comparisons on values of all other types.

As an aside, it is interesting to observe that SQL:2003 included some new material in connection with ORDER BY, allowing the *user* to specify the treatment of NULL, by writing either NULLS FIRST or NULLS LAST. However, no similar addition appears in connection with comparisons.