4 Relational Algebra—The Foundation

4.1 Introduction

The theory book's Chapter 4 describes some operators, as manifested in **Tutorial D**, that together constitute an algebra that is not only *relationally complete* but also irreducibly so (very nearly—apart from `RENAME`, which can be expressed in terms of extension and projection, none of those operators can be discarded without sacrificing completeness). We can use these operators as a basis for testing SQL for relational completeness. If we can show that for every invocation of one of these **Tutorial D** operators there is an equivalent SQL expression, then we will have shown that SQL is relationally complete. By "equivalent" we mean an expression whose table operands are counterparts of the **Tutorial D** relation operands (ignoring the ordering that SQL imposes on the columns) and whose result is a table counterpart of **Tutorial D**'s result, where a table is a counterpart of a relation if and only if it satisfies all of the following conditions:

- every column has a name
- no two distinct columns have the same name
- no row appears more than once
- `NULL` doesn't appear in place of a value anywhere in the table
- every row consists entirely of its column values and doesn't somehow contain any additional data

Strictly speaking, SQL cannot be regarded as relationally complete until it recognizes the existence of relations of degree zero: `TABLE_DEE` and `TABLE_DUM` in **Tutorial D**. Charitably overlooking that omission, we will discover that SQL is otherwise relationally complete, though it wasn't so prior to 1992. However, the table counterparts we will find in SQL will give opportunities for further comment on the language design. We will also discover some of the consequences that arise when SQL's operators are used on tables that do not satisfy all of these conditions.

Figure 4.1, repeated from the theory book, shows the current values of relvars named `IS_CALLED` and `IS_ENROLLED_ON`, which we will now take to be SQL tables (as the current values of SQL base tables).

Please note very carefully that all examples and accompanying explanations assume, unless otherwise stated to the contrary, that the tables involved satisfy the above conditions. Otherwise, as they say, "all bets are off".

Download free eBooks at bookboon.com

IS_CALLED	
StudentId	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

IS_ENROLLED_ON	
StudentId	CourseId
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

Figure 4.1: IS_CALLED and IS_ENROLLED_ON

This will be our running example here too.

In the theory book Example 4.1 is the first example of an invocation of a relational operator, using `JOIN` to derive the `ENROLMENT` relation from `IS_CALLED` and `IS_ENROLLED_ON`. Here it shows an SQL equivalent to `IS_CALLED JOIN IS_ENROLLED_ON`.

Example 4.1: Joining IS_CALLED and IS_ENROLLED_ON in SQL

```
SELECT * FROM IS_CALLED NATURAL JOIN IS_ENROLLED_ON
```

Download free eBooks at bookboon.com

Click on the ad to read more

This is an example of an SQL *table expression*. I have been assuming you are already familiar with the SELECT-FROM-WHERE structure for certain table expressions. Here I give an explanation in a different style from that found in the SQL text books, appealing to the concept of operator invocation that is used in the theory book.

Explanation 4.1

- Example 4.1 is an invocation of the SQL operator SELECT and for that reason I shall refer to such table expressions as SELECT expressions.
- Here the SELECT operator operates on the table denoted by the table expression FROM IS_CALLED NATURAL JOIN IS_ENROLLED_ON, an invocation of the operator FROM. I shall call such table expressions FROM expressions.
- The FROM operator here is operating on the table denoted by the table expression IS_CALLED NATURAL JOIN IS_ENROLLED_ON, an invocation of the operator NATURAL JOIN.
- NATURAL JOIN here is operating on the tables denoted by the table expressions IS_CALLED and IS_ENROLLED_ON, each in turn denoting the table that is the current value of the variable (base table) of that name.
- NATURAL JOIN is almost equivalent to **Tutorial D**'s JOIN. It differs only in being noncommutative because of the ordering to the columns of an SQL table. The common columns appear first in the result, in the order in which they appear in the left operand. Then come the remaining columns of the left operand, followed by the remaining columns of the right operand. As in **Tutorial D**, common columns must be of the same type in both operands.
- FROM is an operator that takes a commalist of one or more table expressions. In this example the list has just one element, IS_CALLED NATURAL JOIN IS_ENROLLED_ON, and the result is that table. An invocation of FROM is usually referred to as a FROM clause. A FROM clause is not permitted to exist in isolation—it must appear in some containing SELECT expression. Similarly, some table expressions are permitted only when they appear as elements of a FROM clause. Simple table names and invocations of NATURAL JOIN are a case in point. The result of a FROM clause must always be operated on by some other clause. In Example 4.1 it is operated on by a SELECT clause. It can also be operated on by any clause that immediately follows it syntactically, such as a WHERE clause, for example. As we shall see, SQL dictates a strict order in which the clauses of a SELECT expression must appear. Evaluation always starts at the FROM clause, then proceeds forwards from clause to clause, then finally back to the SELECT clause.

Download free eBooks at bookboon.com

- SELECT is an operator that takes an explicit or, as in Example 4.1, implicit commalist of column specifications followed by an invocation of FROM. The text from the word SELECT up to, but not including, the word FROM is usually referred to as a SELECT clause. The clauses following the SELECT clause define a table on which the SELECT clause operates. The shorthand SELECT * is equivalent to a commalist specifying each column in turn of the input table defined by the following clauses. Thus, SELECT * FROM *t* is very similar to **Tutorial D**'s identity projection, *t*{ALL BUT}—it yields the table *t*. (When *t* is just a single table name, the shorthand TABLE *t* is available as equivalent to the SELECT expression SELECT * FROM *t*.)

Historical Note:

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.

Figure 4.2 shows the result of evaluating Example 4.1. You can see that it depicts exactly the same table as the current value of ENROLMENT shown in Figure 1.2. Note in particular that the left-to-right order of the columns is as shown in Figure 1.2. When the table operands are reversed the result is the different SQL table depicted in Figure 4.2a.

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 4.2: The result of IS_CALLED NATURAL JOIN IS_ENROLLED_ON

Download free eBooks at bookboon.com

StudentId	CourseId	Name
S1	C1	Anne
S1	C2	Anne
S2	C1	Boris
S3	C3	Cindy
S4	C1	Devinder

Figure 4.2a: The result of IS_ENROLLED_ON NATURAL JOIN IS_CALLED

Effect of NULL

Let *c* be a common column in *t1* NATURAL JOIN *t2*. Then there can be no row in the result for which *c* IS NULL evaluates to TRUE, even if both operands contain such a row. In fact, for each common column *c*, *c*=*c* IS UNKNOWN must evaluate to FALSE for every row of the result. (Recall that *c*=*c* IS UNKNOWN can evaluate to TRUE even when *c* IS NULL does not.)

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.

What if NATURAL JOIN is missing?

In the absence of NATURAL JOIN Example 4.1 has to be replaced by something rather more longwinded, as shown in Example 4.1a.

Example 4.1a: Joining IS_CALLED and IS_ENROLLED_ON in original SQL

```
SELECT  IC.StudentId, Name, CourseId
FROM    IS_CALLED AS IC, IS_ENROLLED_ON AS IE
WHERE   IC.StudentId = IE.StudentId
```

Download free eBooks at bookboon.com

Explanation 4.1a

- The FROM clause now has two elements. When there are two elements, *t1* and *t2,* the result is equivalent to `t1 CROSS JOIN t2`, which is SQL's counterpart of `t1 TIMES t2` in **Tutorial D**. However, TIMES requires its operands to have disjoint headings, whereas CROSS JOIN is defined for all pairs of SQL tables. When *t1* and *t2* each have a column named *c*, the result has two columns named *c.* In general, when *t1* has *m* columns named *c* and *t2* has *n,* `t1 CROSS JOIN t2` has *m+n* columns named *c.*

- Following the FROM clause is a WHERE clause, denoting an invocation of the operator WHERE. The operands are the table resulting from the FROM clause and the condition following the word WHERE. SQL's WHERE operator is equivalent to **Tutorial D**'s operator of the same name when its table operand represents a relation.

- The result of the FROM clause has two columns of the same name, `StudentId.` The condition specified in the WHERE clause uses *range variables,* IC and IE, to distinguish between these two columns. The distinction is possible here, thanks to the fact that the same column name isn't used more than once in either of the two operand tables (as we shall see later, that is a condition that does not always apply, even though the same column name cannot be used more than once in a base table).

- The range variables are defined in the FROM clause alongside the table expressions to which they apply. The key word AS separating the table expression from the range variable name is optional. If the table expression consists of just a table name, unaccompanied by a range variable, then that table name serves also as a range variable name.

- A range variable is so-called because it is considered to "range over" each element in turn of a collection, the collection in the example at hand being the rows of a table. Note carefully that although the expression `IE.StudentId` is a column *reference*, it is not a column *name*. It references a particular column named `StudentId.` The prefix "`IE.`" is required because without it the column reference would be ambiguous.

Historical Notes

You may have learned a different term for *range variable*, which was used by Codd in his early papers but not adopted by the SQL standard until 2003. In some SQL texts it is called *alias* but this is not at all appropriate, really, because that would imply that it is a table name and therefore denotes a table rather than a row. The SQL standard uses the equally inappropriate term *correlation name* (it doesn't denote a correlation, whatever that might be), but only for the case where the name is explicitly given (via AS in the example) and not for the case where a simple table name doubles as a range variable name. In SQL:2003 range variable was adopted as a convenient single term to cover the more general case.

Download free eBooks at bookboon.com

In his seminal 1970 paper [3] E.F. Codd defined the relational algebra, subsequently adopted in various guises by ISBL, Business System 12, and much later, **Tutorial D**. Such an algebra was clearly suited to the conventional style of most computer languages, but also in that paper Codd proposed an alternative approach based more closely on the predicate calculus. He called this notation relational calculus and in a later paper [4] proposed a concrete syntax which he called "Data Sublanguage ALPHA".

It seems that the authors of SEQUEL took inspiration from both relational algebra and Codd's ALPHA, as well as the style of the typical scripting languages of that time that were used for generating reports from files. (The SEQUEL paper explicitly mentions as a source of inspiration such a scripting language, the IBM product GIS—Generalized Information System.)

The idea to use range variables in SEQUEL, thence in SQL, came from ALPHA. Now, when the planners of Business System 12 (of which I was one) were studying Codd's papers, we realized very early on that we needed to choose a style for our language. The algebra appealed because of its conventionality as a set of operators closed over something—relations, of course—and happily the prototype implementation ISBL, which used the algebra defined in reference [14], provided answers to various questions that Codd's proposal had raised, in particular, questions concerning the headings of relations resulting from invocations of these operators.

Download free eBooks at bookboon.com

Click on the ad to read more

ALPHA also had appeal, but a similar question concerning headings arose and this time we could not find a satisfactory answer. Were these "dot qualified" names arising from the use of range variables to be the actual attribute names in the result's heading? If so, what if that relation were input to another ALPHA expression? Did the result of that expression have doubly qualified attribute names? And so on, yielding longer and longer names with more and more dots in them? If so, well, we didn't think that would be acceptable and in any case in those days computer memory was at a premium and for performance reasons we needed reasonably small, fixed-length storage slots to accommodate attribute names. On the other hand, if these qualifiers did not appear in the result headings, how were attributes of the same name to be distinguished? We decided against support for notation based on Codd's calculus, because no implementation had been made to provide answers to these questions. Moreover, if and when satisfactory answers appeared, then we would have the option to provide such support as a mapping to the algebra-based language—an alternative interface for users who might prefer that style.

The authors of SEQUEL decided differently. In fact they decided to be different from both the algebra and the calculus. They decided that in a base table each column would have a unique name but that requirement would not carry through in general to results of table expressions. The presence of two or more columns with the same name didn't matter much in SEQUEL, or in SQL prior to 1992, because table expressions other than simple table names were not permitted to appear as operands of `FROM`—the language was thought by some, erroneously as it turned out, to be relationally complete even with that restriction. A similar remark applies in connection with columns that have no name at all, as in `SELECT SUM(X) FROM T`, for example. If tables with such columns can arise only as results of `SELECT` expressions, and `SELECT` expressions aren't permitted in `FROM` clauses, then there's never any possibility to reference such columns by name, and if the language is complete, then there isn't any need to either. The reason why relational completeness in SQL requires support for SELECT expressions in the FROM clause is given in Chapter 5, Section **5.6, Summarization in SQL.**

Another point motivating the decisions taken in ISBL and BS12 was that it was felt that every expression denoting a relation should be assignable to a relation variable of the same type. SEQUEL was silent on the subject of variables but in SQL not every table expression can be the value of a base table, because a base table is required to have a unique name for every column.

As for the strange restrictions that govern the syntax of `SELECT` expressions, requiring a query to be expressed as a fixed sequence of invocations of specific operators, that would have been presumably inspired by GIS. The typical style of report generation languages in those days was based around something like this, assuming for simplicity that a single input file contains all the data needed for the report:

1. Specify the input file, whose physical layout in terms of fields in records would be described in something called the Data Dictionary, which gave names to those fields. So this became SEQUEL's `FROM` clause.

Download free eBooks at bookboon.com

2. Specify a condition to select the desired records from that file. So that became the `WHERE` clause.

3. Specify the values, derived from the selected records, that were to appear in the output report. So that became the `SELECT` clause, placed first in spite of it being actually the last step, to align the complete expression with the normal style of English sentences.

The foregoing account is largely conjecture but it seems quite reasonable from the evidence available. Recalling that the "S" in SEQUEL and SQL originally stood for "structured" (though this latter never became official), we can add the observation that "structured" was something of a buzz word in the 1970s, when the discipline of structured programming rightly became fashionable. However, there is of course no connection at all between the structure of SQL table expressions and the discipline of structured programming.

4.2 Relations and Predicates

Sections 4.2 in the theory book states the importance of *the closed world assumption* and applies just as well to SQL, insofar as it can be deemed applicable in the presence of that intrusive truth value, `UNKNOWN`, discussed in Chapter 3.

4.3 Relational Operators and Logical Operators

Section 4.3 in the theory book prepares the ground for subsequent sections in which each specific relational operator is paired with its logical counterpart, such that, for example, *`r1` `JOIN` `r2`* denotes the relation representing the extension of the predicate *`p1` `AND` `p2`*, the conjunction of the predicates for the operand relations. It follows that where we can find SQL counterparts of those relational operators, invocations of those counterparts will in turn represent extensions of the predicates given in the theory book.

The definitions that appear in the following sections use the following conventions:

- Symbols beginning with *t* denote tables and those beginning with *r* denote rows. In the theory book, of course, it's the other way around, so to speak, because *r* stands for relation, *t* for tuple.

- Because the columns of a table, and therefore the components of a row, are ordered in SQL, the term *concatenation* is used in its usual sense to refer to connecting two or more things (headings or rows), one after the other in the order specified.

Download free eBooks at bookboon.com

4.4 JOIN and AND

In the theory book Section 4.4 is all about one operator, `JOIN`. SQL's closest counterpart, `NATURAL JOIN`, has already been covered. Here we look at several other "join" operators defined in SQL. We don't really need to, as `NATURAL JOIN`, if considered as primitive, renders all the others redundant as shorthands. But as has already been mentioned, you won't find `NATURAL JOIN` in every SQL product. `CROSS JOIN` has already been mentioned as the operator implicitly used in joining the tables specified in a `FROM` clause's commalist. We start by giving its full definition.

Definition of `CROSS JOIN`

> Let *s* = *t1* **`CROSS JOIN`** *t2*, where *t1* and *t2* are table expressions optionally accompanied by range variables. Then:
>
> - The heading *Hs* of *s* is the concatenation of the headings of *t1* and *t2*, in that order.
> - If *r1* is a row appearing *n1* times in *t1* and *r2* is a row appearing *n2* times in *t2*, then the row formed by concatenation of *r1* and *r2* in that order appears *n1*n2* times in the body of *s*.

It follows that the degree of the result is the sum of the degrees of the operands and its cardinality is the product of their cardinalities, as with *`r1 TIMES r2`* in **Tutorial D**.

Download free eBooks at bookboon.com

Click on the ad to read more

Interesting properties of `CROSS JOIN`

Compare these with the "interesting properties of `JOIN`" given in the theory book.

`CROSS JOIN` is *associative* but not *commutative*, for the reason given in Section 4.1.

Unlike `JOIN` and `NATURAL JOIN`, `CROSS JOIN` is not *idempotent*. Let *t* be any table and let *n* be its cardinality. Then *t* `CROSS JOIN` *t* has twice as many columns as *t* and n^2 rows.

Also unlike `JOIN`, `CROSS JOIN` has no identity value. If *t* is a table, there is no table *t0* such that *t* `CROSS JOIN` *t0* = *t*. That's because *t0* would be required to have no columns and exactly one row, but SQL doesn't recognize the existence of tables with no columns.

We can now define `FROM` in terms of `CROSS JOIN`.

Definition of `FROM`

Recall that the operand of `FROM` is denoted by a commalist, each element of that commalist being a table expression optionally accompanied by a range variable name. Then we can write:

`FROM` *fe*$_1$, *fe*$_2$, … *fe*$_n$ ($n \geqslant 0$) is equivalent to `FROM` *fe*$_1$ `CROSS JOIN` *fe*$_2$ `CROSS JOIN` … `CROSS JOIN` *fe*$_n$.

Recall that the operands of `CROSS JOIN` can also include range variables. Note also that an invocation of `CROSS JOIN` is itself a table expression and can thus appear in an element of a `FROM` clause. In the light of that observation, consider Example 4.1b

Example 4.1b: FROM giving indistinguishable columns of same name

```
FROM   (IS_CALLED AS IC CROSS JOIN IS_ENROLLED_ON AS IE) AS CJ
WHERE  ??.StudentId = ??.StudentId
/* The StudentId columns can no longer be distinguished */
```

The only qualifier available for the columns of the result of that `FROM` clause is `CJ`. The range variables `IC` and `IE` are rendered out of scope by the definition of `CJ`.

Download free eBooks at bookboon.com

Historical Notes

A restricted version of FROM was defined in SEQUEL and thence in the first implementations of SQL. The restriction was that each operand had to be a simple table name, referencing either a base table or a "view" (SQL's counterpart of **Tutorial D**'s virtual relvars) rather than a table expression of arbitrary complexity. This restriction, along with others concerning the table expressions on which views were defined, is one of those that rendered SEQUEL and early SQL relationally incomplete. See Chapter 5, Section **5.6, Summarization in SQL**.

CROSS JOIN was added to the international standard in 1992, along with the other explicit JOIN operators. It is completely redundant and can hardly be regarded as a shorthand, but perhaps its use makes certain expressions clearer than they would otherwise be. It remains an optional conformance feature.

Another variety of JOIN is illustrated in Example 4.1c—the "named columns join". Here the common columns to be used for matching rows are specified explicitly in a parenthesized commalist following the word USING. As with NATURAL JOIN, each column used for matching appears just once in the result, so Example 4.1c is in fact equivalent to Examples 4.1 and 4.1a.

Example 4.1c: Obtaining a natural join by specifying the common columns

```
SELECT * FROM IS_CALLED JOIN IS_ENROLLED_ON USING ( StudentId )
```

However, a named columns join doesn't always have an equivalent formulation using NATURAL JOIN. That's because although each USING column must be a common column, it is not necessary to specify *all* the common columns. A common column whose name does not appear in the USING list gives rise to two columns of that name in the result, which therefore does not represent a relation.

To cater for cases where the columns used for matching purposes are not common columns, and/or the matching operator is not "=", the required joining condition can be spelled out as shown in Example 4.1d, in parentheses following the key word ON.

Example 4.1d: Explicitly specifying the join condition

```
SELECT *
FROM   IS_CALLED JOIN IS_ENROLLED_ON
       ON ( IS_CALLED.StudentId = IS_ENROLLED_ON.StudentId )
```

Note carefully that Example 4.1d is *not* equivalent to Example 4.1. That's because the result now contains two StudentId columns, as would of course be required if they didn't have the same name. In fact, Example 4.1c is equivalent to the table expression obtained by replacing JOIN by a comma and ON by WHERE.

Download free eBooks at bookboon.com

Now, the key word `JOIN` in all of the foregoing examples can be harmlessly preceded by the word `INNER`. SQL also supports what are called "outer joins". The outer join of *t1* and *t2* contains all the rows of the inner join and possibly some more if either operand has rows which fail to participate in the inner join. Such a row might participate in the outer join, accompanied by `NULL` for each column of the other operand. The key words `LEFT`, `RIGHT`, and `FULL`, each optionally followed by `OUTER`, are used to specify whether unmatched rows of the first (left) operand, the second (right) operand, or both operands, respectively, are to appear in the result. Example 4.1e shows an SQL outer join. As well as the rows shown in Figure 4.2, a single row for student S5 appears in the result, with `NULL` in place of a value for `CourseId`.

Example 4.1e: An SQL outer join

```
SELECT *
FROM   IS_CALLED NATURAL LEFT JOIN IS_ENROLLED_ON
```

Note that adding `LEFT` to an invocation of `CROSS JOIN` has no effect unless the right-hand operand table is empty. As outer joins in general denote tables that are not counterparts of relations, they merit no further discussion here.

Download free eBooks at bookboon.com

Click on the ad to read more

Historical Notes

Apart from NATURAL, CROSS, and FULL, all the JOIN options described in this section became mandatory conformance features in SQL:1999.

When outer joins were added to SQL in 1992, the use of NULL for the noncommon columns in unmatched rows was a fairly obvious choice. The designers of Business System 12 also recognized the requirement for outer joins but could not use NULL for the same purpose because the decision had already been taken not to include any such construct in that language. Instead, they defined an operator, MERGE, similar to SQL's LEFT JOIN but differing from LEFT JOIN in two respects. First, the values to be used for the noncommon attributes of unmatched tuples had to be explicitly specified in the invocation (and they had to be values, of course—there was no such thing as NULL). Secondly, the common attributes were required to constitute a superkey of the second operand, thus guaranteeing that the join was many-to-one or one-to-one (i.e., each tuple of the first operand would be joined with at most one tuple of the second). In Example 4.1e the join is one-to-many, which makes the appearance of just one row for student S5 seem somewhat arbitrary. The superkey requirement did not further restrict the second operand in any way because BS12 required a key to be specified for every base relvar and, furthermore, from these declared keys, and the semantics of each relational operator, BS12 was able to infer keys for the results of invocations of relational operators.

4.5 Renaming Columns

SQL has no direct counterpart of RENAME. To derive the table on the right in Figure 4.4 from the table on the left, **Tutorial D** has `IS_CALLED RENAME { StudentId AS Sid }`.

StudentId	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

Sid	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

Figure 4.4: Tables differing only in a column name

Example 4.2 shows how the same effect can be achieved in SQL. Note that the SELECT clause has to include all the columns that are not subject to renaming, as well as those that are.

Example 4.2: Renaming a column

```
SELECT T1.StudentId AS Sid1, T1.Name, T2.StudentID AS Sid2
FROM   IS_CALLED T1, IS_CALLED T2
WHERE  T1.Name = T2.Name
  AND  T1.StudentId < T2.StudentId
```

Download free eBooks at bookboon.com

Explanation 4.2

- Each element of a SELECT clause is an expression of arbitrary complexity that can reference one or more columns of the SELECT clause's input table. Note that the expression is not *required* to reference any columns: it might be just a literal, for example.
- If the expression consists of just a column reference, then the name of the referenced column is the name of the corresponding column in the result.
- If the expression is followed by the key word AS, then the resulting column has the column name given after that key word.

SQL allows the result to have two or more columns of the same name. For example, if we replace Sid by Name in Example 4.2, we still have a valid SQL SELECT expression.

Historical Notes

Column renaming with AS wasn't in SEQUEL or the early implementations of SQL. It was added to the international standard in 1992, along with the liberation of FROM to allow table expressions in general as operands. Without AS, some columns in the result of FROM would have to be anonymous or have nonunique names. Such columns cannot be referenced in subsequent clauses such as WHERE and SELECT.

The fact that, in standard SQL (though not in all implementations), AS can assign the same name to more than one column in the same SELECT clause may surprise you. The rationale for this is that prior to 1992 an SQL table expression could already result in a table with two or more columns of the same name (SELECT C, C FROM T, for example, or SELECT T1.C, T2.C FROM T1, T2). A prohibition on the use of AS for this purpose would be futile unless duplicate column names were outlawed altogether, which was out of the question for the paramount reason of backwards compatibility.

Using RENAME in combination with JOIN

Example 4.3 in the theory book gives pairs of ids of students having the same name, by joining two renamings of IS_CALLED. Example 4.3a gives an equivalent expression in SQL.

Example 4.3a: Renaming and joining

Student *Sid1* is called *Name* and so is student *Sid2*

```
SELECT *
FROM   (SELECT StudentId AS Sid1, Name FROM IS_CALLED)
       NATURAL JOIN
       (SELECT StudentId AS Sid2, Name FROM IS_CALLED)
```

Download free eBooks at bookboon.com

As before, the result sagely tells us that student S1 (Anne) has the same name as herself and also shows two pairings of S1 with S5 (both named Boris). The pairing of a student id with itself can be avoided by adding `WHERE Sid1 <> Sid2` to the `WHERE` clause. The duplicate pairings can further be avoided by using < instead of <> in this addition, but that trick assumes that an ordering is defined for type `SID`, which is not necessarily the case. If `NATURAL JOIN` is not available, an expression such as the one shown in Example 4.3b could be used instead.

Example 4.3b: Renaming and joining without NATURAL JOIN

Student *Sid1* is called *Name* and so is student *Sid2*

```
SELECT T1.StudentId AS Sid1, T1.Name, T2.StudentID AS Sid2
FROM   IS_CALLED T1, IS_CALLED T2
WHERE  T1.Name = T2.Name
   AND T1.StudentId < T2.StudentId
```

4.6 Projection and Existential Quantification

Intuitively it might seem that projection in SQL is simply a matter of specifying the required columns in the `SELECT` clause, as in Example 4.4a.

LIGS University
based in Hawaii, USA
is currently enrolling in the Interactive Online BBA, MBA, MSc, DBA and PhD programs:
- enroll by October 31st, 2014 and
- save up to 11% on the tuition!
- pay in 10 installments / 2 years
- Interactive Online education
- visit www.ligsuniversity.com to find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education.
More info here.

Download free eBooks at bookboon.com

Click on the ad to read more

Example 4.4a: Projection (incorrect)

Student *StudentId* is enrolled on some course.

```
SELECT StudentId FROM IS_ENROLLED_ON
```

Unfortunately, though, if some student is currently enrolled on more than one course (as indeed student S1, Anne, is in our example database), then the row giving that student's id appears twice in the result, which because of that duplicate appearance does not represent a relation. To avoid multiple appearances of the same row SQL requires you to write the word `DISTINCT`, as in Example 4.4b. (The key word `ALL` can be given instead of `DISTINCT`, clarifying that duplicate rows are not to be eliminated. As `ALL` is the default option, it is rarely seen in examples and I do not use it in this book.)

Example 4.4b: Projection (correct version)

Student *StudentId* is enrolled on some course.

```
SELECT DISTINCT StudentId FROM   IS_ENROLLED_ON
```

In more complicated examples it is sometimes quite difficult to tell whether omission of `DISTINCT` would give rise to duplicate rows. It might therefore seem good advice to always write `DISTINCT`. Indeed, I would certainly advocate such practice to students having to write SQL expressions in solutions to questions in exam papers, for example, but if it were followed blindly in commercial systems, then many queries would run very much slower than need be because typical SQL implementations have little or nothing in the way of built-in intelligence to recognize cases where duplicate rows cannot possibly arise. Although such intelligence is quite feasible within acceptable limits (and was used in Business System 12, for example), the inclusion of `DISTINCT` allows SQL implementations to place the responsibility for duplicate elimination on the user.

Recall that **Tutorial D** allows projection to be expressed either by listing the attributes to be included or by listing the ones to be excluded, using `ALL BUT`. SQL has no counterpart of the `ALL BUT` variety.

Effect of `NULL`

Rows *r1* and *r2* are considered as duplicates in SQL when *r1* `IS NOT DISTINCT FROM` *r2* evaluates to `TRUE`. Recall that *r1* = *r2* can evaluate to `UNKNOWN` in such cases. So `SELECT DISTINCT` is one of those exceptional cases where, in the words of the SQL standard, "multiple null values are treated together".

Download free eBooks at bookboon.com

Historical Notes

The requirement for the user to specify whether duplicate rows are to be eliminated was in original SQL. In fact it was also in SEQUEL but with an interesting difference: elimination of duplicates was the default option in SEQUEL.

It might seem a simple matter to add support for column exclusion to SQL by syntax such as `SELECT * EXCEPT <column name list>`, but a proposal to that effect for the SQL standard was rejected in 2004 as a result of objections raised by the USA delegation. They argued that the use of `SELECT *` is frowned upon by most SQL experts and should be discouraged. To that end, they would accept no enhancements to `SELECT *`.

Of course, the reason why `SELECT * FROM T` is so deprecated is that its meaning changes whenever the definition of `T` is changed—for example by use of `ALTER TABLE`. Applications that always spell out exactly which columns are required in the `SELECT` clause can be to a greater extent immune to such schema changes. (A similar comment would apply to the use of **Tutorial D** expressions such as `IS_ENROLLED_ON` and `IS_ENROLLED_ON{ALL BUT CourseId}`, but the problem is addressed in **Tutorial D** by allowing an application to define its own perception of the database using a special form of local relvar, not mentioned in the theory book.) However, in SQL the columns of a table expression might be explicitly spelled out in the `FROM` clause, as in Example 4.3a, in which case the use of `SELECT *` is both convenient and harmless.

How `ENROLMENT` was split

Example 4.5 in the theory book shows how relvars `IS_CALLED` and `IS_ENROLLED_ON` can be derived from the original `ENROLMENT` relvar, using projection in the initial assignment to those relvars. Here is how the same effect can be achieved in SQL:

Example 4.5: Splitting `ENROLMENT`

```
CREATE TABLE IS_CALLED
AS (SELECT DISTINCT StudentId, Name FROM ENROLMENT)
WITH DATA ;

ALTER TABLE IS_CALLED ADD CONSTRAINT PRIMARY KEY ( StudentId ) ;

CREATE TABLE IS_ENROLLED_ON
AS (SELECT DISTINCT StudentId, CourseId FROM ENROLMENT)
WITH DATA ;

ALTER TABLE ADD CONSTRAINT PRIMARY KEY ( StudentId, CourseId ) ;

DROP TABLE ENROLMENT ;
```

Download free eBooks at bookboon.com

Explanation 4.5:

- **`CREATE TABLE IS_CALLED`** announces that what follows defines a base table named `IS_CALLED.`
- **`AS (SELECT DISTINCT StudentId, Name FROM ENROLMENT)`** specifies that the columns of `ENROLMENT` and their declared types are as in the specified expression.
- **`WITH DATA`** additionally specifies that the table resulting from the specified expression is to be the initial value of `IS_CALLED.`
- **`ALTER TABLE IS_CALLED ADD PRIMARY KEY ( StudentId )`** specifies a constraint to the effect that no two distinct rows having the same `StudentId` value can ever appear simultaneously in `IS_CALLED.` Note that this constraint has to be given as a separate statement from the one that creates the base table. If the key word `DISTINCT` had been omitted, the `CREATE TABLE` statement would have succeeded but the `ALTER TABLE` statement would have failed because the required constraint would have been violated by the two appearances of the row for student S1, Anne.
- Similar comments apply to the `CREATE` and `ALTER TABLE` statements for `IS_ENROLLED_ON`, but in the equivalent example in the theory book I noted that the specification `KEY {StudentId, CourseId}`, required by **Tutorial D**, is theoretically redundant because the entire heading is always a superkey. Here, the corresponding `ALTER TABLE` statement is not redundant because in the absence of any key constraints SQL allows the same row to appear several times simultaneously in the same base table.
- **`DROP TABLE ENROLMENT`** destroys the variable we have no further use for.

Two special cases of projection

In the theory book this section describes the identity projection, *r* `{ ALL BUT }`, and the projection on no attributes, *r* `{ }`, which yields `TABLE_DUM` when *r* is empty, otherwise `TABLE_DEE`. As we have already seen, the identity projection is represented in SQL by `SELECT * FROM` *t*, but SQL does not recognize the existence of columnless tables and so `SELECT FROM` *t* is not supported.

Download free eBooks at bookboon.com

Historical Notes

The history surrounding `CREATE TABLE ... AS ...` is given in Chapter 2, in the Historical Notes for Section 2.11.

Various attempts have been made to develop "natural language query" products, at least in prototype form, whereby queries expressed in a human language are translated into SQL. In the 1980s I became peripherally involved in one such product. Its developers told me that they could handle questions such as "Did any student score more than 90 in the relational database theory exam?" but the translation to SQL was awkward because they had to make a special case for the generation of the `SELECT` clause, which would otherwise include no columns! The **Tutorial D** expression for such a query would end with a projection on no attributes, yielding either `TABLE_DEE` (meaning "yes") or `TABLE_DUM` ("no").

4.7 Restriction and AND

Restriction in **Tutorial D** is available via the `WHERE` operator, and so it is in SQL—we have already seen it several times in this chapter, such as Example 4.3b. However, the subject is introduced in the theory book by Example 4.6, showing how a certain simple restriction can be expressed using `JOIN` and a relation literal. It is useful to show SQL's counterpart of that example, giving the student ids of students named Boris.

Download free eBooks at bookboon.com

Click on the ad to read more

Example 4.6: joining with a table literal

```
SELECT DISTINCT StudentId
FROM   IS_CALLED NATURAL JOIN ( VALUES ( 'Boris' ) ) AS T(Name)
```

As noted in Chapter 2, the columns of a table literal in SQL are anonymous. When a table literal is an operand in a table expression the only way of assigning names to its columns is by giving them in the definition of a range variable—`AS T(Name)` in the example. Example 4.7 shows the equivalent, more intuitive formulation using `WHERE`.

Example 4.7: Restriction in SQL

```
SELECT DISTINCT StudentId
FROM    IS_CALLED
WHERE   Name = 'Boris'
```

The `WHERE` clause operates on the result of the `FROM` clause in analogous fashion to the way it operates on an arbitrary relation expression in **Tutorial D**. `Name = NAME ( 'Boris' )` is an open expression, as defined in the theory book. The key word `DISTINCT` here is redundant, as it happens, because `StudentId` is a declared key for `IS_CALLED`.

Example 4.8 in the theory book finds students whose names start with B. Example 4.8 here shows the SQL counterpart, this being one that cannot feasibly be expressed by joining with a table literal.

Example 4.8: A more useful restriction

```
SELECT *
FROM    IS_CALLED
WHERE   SUBSTRING(Name FROM 1 FOR 1) = 'B'
```

Note in passing the unconventional syntax used to invoke the built-in `SUBSTRING` operator. An alternative way of expression the `WHERE` condition, and one that has been in SQL for longer than `SUBSTRING`, which first appeared in SQL:1992, is `Name LIKE 'B%'`, in which the % character is used as a "wild card" signifying "anything can appear here, even nothing at all"—nowadays, in search engines and the like, * is more commonly used for such purposes.

Download free eBooks at bookboon.com

Effects of `NULL`

A note of caution needs to made here. Recall that in general a conditional expression in SQL can evaluate to `UNKNOWN` as well as `TRUE` or `FALSE`. A row satisfies the `WHERE` condition *c* if and only if *c* is `TRUE`. This is inconsistent with SQL's treatment of database constraints, which are deemed to be satisfied whenever they evaluate to either `TRUE` or `UNKNOWN`. Consider, for example, a constraint declared with condition `NOT EXISTS ( SELECT * FROM T WHERE X <= Y )`, requiring every row in `T` to satisfy the condition `X > Y`. Then it can happen that `SELECT * FROM T WHERE X > Y` is empty—for example, if `T` is not empty but `X IS NULL` is true for every row—even though `SELECT * FROM T` is not empty!

Historical Note

The `SUBSTRING` operator was added to SQL in 1992 and became a mandatory conformance feature in SQL:1999. The motivation behind the unconventional syntax for invoking `SUBSTRING` and other built-in operators lay in a desire to distinguish invocations of built-in operators syntactically from those of user-defined operators.

4.8 Extension and AND

The theory book gives the following simple example of relational extension in **Tutorial D**:

```
EXTEND IS_CALLED : { Initial := FirstLetter ( Name ) }
```

Assuming the user-defined operator FirstLetter is available to the SQL user, this can be expressed easily in SQL but there is a strange quirk in the grammar at play here:

```
SELECT IC.*, FirstLetter ( Name ) AS Initial
FROM   IS_CALLED AS IC
```

Note very carefully that we have to qualify the * using the range variable, `IC`, which in this case ranges over the rows of the current value of `IS_CALLED`. When we use a `SELECT` clause to "add columns" to the table on which it operates, it seems obvious to write * to specify that every column of that operand table is required and to follow it with a commalist of expressions denoting the additional columns. For some reason the official SQL grammar does not allow additional columns to accompany an unadorned *. When the `FROM` clause contains several entries, pure extension becomes more difficult to express in SQL. For example, if the `FROM` clause defines range variables `T1`, `T2`, and `T3`, we could write `SELECT T1.*, T2.*, T3.* ...`, but that would defeat the purpose. To be able to write just a single * to denote all the columns of the `FROM` table, we would have to resort to something like

```
SELECT T.*, …
FROM   ( SELECT * FROM T1, T2, T3 WHERE … ) AS T
```

Download free eBooks at bookboon.com

Quirks like this in the SQL grammar serve only to further exacerbate the difficulties for teachers and students caused by the language's major departures from the theory that I have already described.

Effects of `NULL`

In general, if some argument to a read-only operator invocation is `NULL`, then so is the result of that invocation. For example, `X + Y` evaluates to `NULL` if either `X IS NULL` or `Y IS NULL` evaluates to `TRUE`. There are some exceptions, `IS NULL` being an obvious one. User-defined operators, of course, can make their own arrangements—my putative `FirstLetter` operator, for example, might be defined to return the empty string, `''`, when its argument is `NULL`.

Download free eBooks at bookboon.com

Click on the ad to read more

Historical Notes

Codd did not include an extension operator in his algebra. The reason he gave for this omission did not stand up to scrutiny. He regarded the ability to compute such "additional values" as the responsibility of the application program, using facilities of the host language rather than his proposed "data sublanguage". Database practitioners knew, of course, that this approach was hopelessly flawed and the designers of ISBL and SEQUEL both came up with fairly obvious solutions to rectify matters. ISBL being based firmly on explicit relational operators, it was a simple matter to invent extension. It was no doubt equally easy for SEQUEL to allow "additional columns" to be specified in the `SELECT` clause, but their decision not to require or even allow the user to provide names for those columns should surely have been brought into question by the designers of SQL.

As for the quirk concerning the use of *, a proposal to allow an unadorned * to be accompanied by additional entries in the `SELECT` clause was submitted in 2004 (along with the previously mentioned proposal to support `* EXCEPT`) but was rejected in the face of the same objection (use of * is deprecated and should not be encouraged by the introduction of further enhancements).

4.9 UNION and OR

SQL supports `UNION` explicitly but differently from the way it supports `JOIN` explicitly. As we have seen, `JOIN` is used exclusively within the `FROM` clause, such that `IS_CALLED NATURAL JOIN IS_ENROLLED_ON`, for example, can be an element of that clause but cannot stand alone as a table expression. Instead, `UNION` always connects table expressions that *can* stand alone, these being:

- `SELECT` expressions
- `TABLE` *tn,* which is equivalent to `SELECT * FROM` *tn,* where *tn* is a table name
- `VALUES` expressions
- Invocations of `UNION, INTERSECT` (see Chapter 5, Section **5.2, Semijoin and Composition**) and `EXCEPT` (see section **4.10, Semidifference and NOT**)

Actually, just as SQL has several varieties of `JOIN`, it also has several varieties of `UNION`, none of which is equivalent to the relational operator of that name. The closest approximation to relational union is illustrated in Example 4.9a, a translation to SQL of the theory book's Example 4.9.

Download free eBooks at bookboon.com

Example 4.9a: SQL's closest approximation to relational union

```
SELECT StudentId
FROM   IS_CALLED
WHERE  Name = 'Devinder'
UNION DISTINCT CORRESPONDING
SELECT StudentId
FROM   IS_ENROLLED_ON
WHERE  CourseId = 'C1''
```

The key word `DISTINCT` is optional and implied by default (somewhat curiously so, considering that its opposite, `ALL`, is the default option in the `SELECT` clause). It specifies that no row is to appear more than once in the result. Thus, there is never a need to include `DISTINCT` in either of the `SELECT` clauses, and this would be the case even if the `WHERE` clause were omitted from the specification of the second operand in Example 4.9a, allowing the same `StudentId` value to appear more than once in that operand.

The key word `CORRESPONDING` specifies that operand columns are to be paired by name, just as in relational union. Thus, the slightly revised version shown in Example 4.9b is also legal and is in fact equivalent to Example 4.9a. Curiously, the corresponding columns do not have to be of the same type! However, each value appearing in a corresponding column of the second operand must be "castable" to a value in the type of its counterpart in the first operand. For example, the character string "+123" is castable to the value 123 of type `INTEGER` but the character string "123.5" is not.

Example 4.9b: Union applied to disparate operands

```
SELECT *
FROM   IS_CALLED
WHERE  Name = 'Devinder'
UNION DISTINCT CORRESPONDING
SELECT *
FROM   IS_ENROLLED_ON
WHERE  CourseId = 'C1'
```

This is legal and equivalent because `CORRESPONDING` specifies that *only* the common columns of each operand are to appear in the result. In this case `StudentId` is the only common column, of course. As usual, the common columns of one operand must be of the same type as their counterparts in the other operand. And as you might expect by now, there has to be at least one common column (SQL doesn't recognize the existence of columnless tables).

Download free eBooks at bookboon.com

A further variation, also equivalent to Example 4.9a, is shown in Example 4.9c.

Example 4.9c: Specifying the columns required

```
SELECT *
FROM   IS_CALLED
WHERE  Name = 'Devinder'
UNION DISTINCT CORRESPONDING BY (StudentId)
SELECT *
FROM   IS_ENROLLED_ON
WHERE  CourseId = 'C1'
```

The required columns can be specified explicitly in a parenthesized commalist following the key word BY. Columns thus specified must be common to both operands but the list is not required to specify *all* the common columns.

I turn now to the use of UNION without CORRESPONDING. Example 4.9d is derived from 4.9a merely by omitting CORRESPONDING and is in fact equivalent to 4.9a, but only because the operands have identical SELECT clauses.

Download free eBooks at bookboon.com

Click on the ad to read more

Example 4.9d: UNION without CORRESPONDING

```
SELECT StudentId
FROM   IS_CALLED
WHERE  Name = 'Devinder'
UNION DISTINCT
SELECT StudentId
FROM   IS_ENROLLED_ON
WHERE  CourseId = 'C1'
```

When `CORRESPONDING` is omitted, names are not used at all in the pairing of columns. Instead, SQL's definition, in yet another departure from relational database theory, depends on an ordering of the columns: the first column of the first operand is paired with the first column of the second operand, the second with the second, and so on. As with `CORRESPONDING`, columns thus paired do not have to be of the same type. Furthermore, the two operand tables must have the same number of columns, so that there is no unpaired column in either operand, also as in relational union.

Although the operand columns in 4.9d still have the same name, `StudentId`, that is not a requirement in this variety of `UNION`. For example, `SELECT StudentId AS X` could be the `SELECT` clause of the second operand. However, if corresponding columns do not have the same name, then the corresponding column in the result is effectively anonymous (the standard defines it to have an unpredictable system-generated name). Actually, some implementations use the column names of the first operand here, thus destroying the normal commutativity of `UNION`. The user of an implementation that strictly follows the standard would perhaps be well advised always to make sure the corresponding columns have the same name anyway, to avoid the unpredictability of system-generated names and to improve portability from one implementation to another.

Further varieties of `UNION` arise when we replace the key word `DISTINCT` by `ALL` in any of the foregoing examples, as in Example 4.9e. `ALL` specifies that if row *r* appears *n* times in one operand and *m* times in the other, then it appears *n*+*m* times in the result—*i.e.,* no elimination of duplicate rows takes place.

Example 4.9e: UNION ALL

```
SELECT StudentId
FROM   IS_CALLED
WHERE  Name = 'Devinder'
UNION ALL
SELECT StudentId
FROM   IS_ENROLLED_ON
WHERE  CourseId = 'C1''
```

Download free eBooks at bookboon.com

Clearly, `UNION ALL` represents another departure from relational theory. However, it is commonly used when the operands can be guaranteed to be disjoint because in such cases omission of `ALL` would incur the possibly significant overhead of the duplicate elimination process with no effect on the final result.

Some authorities have argued that there really ought to be yet another variety of `UNION`, such that if row *r* appears *n* times in one operand and *m* times in the other, with $m \geqslant n$, then it appears *m* times in the result. Relational devotees might smile at this observation but refrain from comment.

Effect of `NULL`

In the case of *`t1`* `UNION DISTINCT` *`t2`*, row *r* appears in the result, just once, if and only if either *`t1`* or *`t2`* contains at least one appearance of row *s* such that *r* `IS NOT DISTINCT FROM` *s* evaluates to `TRUE`. In other words, `NULL` is treated as equal to itself for the purposes of duplicate elimination.

Historical Notes

The grammar given in the SEQUEL paper uses the mathematical symbol ∪ for union and defines just this single version. The body of the paper gives only a brief mention of this operator, apparently implying that its usual mathematical definition is intended.

Original SQL included just `UNION` and `UNION ALL`. `CORRESPONDING` was added in 1992 but remains an optional feature. Support for using the key word `DISTINCT` instead of just omitting `ALL` was added in SQL:1999 but remains an optional feature.

4.10 Semidifference and NOT

In this section in the theory book I first describe the relational difference operator, named `MINUS` in **Tutorial D**. Example 4.10 here shows SQL's closest counterpart of that operator.

Download free eBooks at bookboon.com

Example 4.10: Difference in SQL

```
SELECT StudentId
FROM   IS_CALLED
WHERE  Name = 'Devinder'
EXCEPT DISTINCT CORRESPONDING
SELECT StudentId
FROM   IS_ENROLLED_ON
WHERE  CourseId = 'C1'
```

The syntax for `EXCEPT` exactly parallels that for `UNION`. The key words `DISTINCT`, `ALL`, and `CORRESPONDING` have exactly the same significance as in `UNION`, and `DISTINCT` remains the default option. When `CORRESPONDING` is not given, columns are paired by ordinal position, as in `UNION`.

Download free eBooks at bookboon.com
Click on the ad to read more

t1 EXCEPT DISTINCT *t2* returns the table consisting of a single appearance of each row that appears in *t1* but not in *t2*. *t1* EXCEPT ALL *t2* returns the table consisting of $n-m$ appearances of each row that appears n times in *t1* and m times in *t2*, with $n>m\geqslant 0$.

Thanks to the implicit exclusion of noncommon attributes, EXCEPT CORRESPONDING can also sometimes be used to obtain a semidifference (*r1* NOT MATCHING *r2* in **Tutorial D**), but only when every column of the first operand is a common column. That is not the case in the theory book's Example 4.11, IS_CALLED NOT MATCHING IS_ENROLLED_ON. In general, therefore, semidifference needs to be expressed in a more elaborate longhand in SQL. Example 4.11a does it by joining the result of Example 4.10 with IS_CALLED. Example 4.11b does it by using SQL's comparison operator NOT IN, meaning "is not a member of", in a WHERE condition. Example 4.11c

Example 4.11a: Semidifference via EXCEPT and JOIN

```
SELECT *
FROM    (SELECT StudentId
         FROM    IS_CALLED
         WHERE   Name = 'Devinder'
         EXCEPT DISTINCT CORRESPONDING
         SELECT StudentId
         FROM    IS_ENROLLED_ON
         WHERE   CourseId = 'C1') AS T1
         NATURAL JOIN IS_CALLED
```

Example 4.11b: Semidifference via NOT IN and a subquery

```
SELECT StudentId
FROM    IS_CALLED
WHERE   Name = 'Devinder'
  AND   StudentId NOT IN (SELECT StudentId
                          FROM    IS_ENROLLED_ON
                          WHERE   CourseId = 'C1')
```

Example 4.11c: Semidifference via "quantified comparison" and a subquery

```
SELECT StudentId
FROM    IS_CALLED
WHERE   Name = 'Devinder'
  AND   StudentId <> ALL (SELECT StudentId
                          FROM    IS_ENROLLED_ON
                          WHERE   CourseId = 'C1')
```

Download free eBooks at bookboon.com

The NOT IN expression in Example 4.11b appears to be testing for the appearance of a character string in a table, but in fact the first operand *in this context* is short for ROW(StudentId)—and recall from Chapter 2 that the key word ROW is optional, even when the row to be specified has more than one column value. Thus the expression tests for the nonappearance of a given row in a given table. (You won't be surprised to hear that if the word NOT is omitted, then the expression becomes a test for appearance rather than nonappearance.)

In Example 4.11c <> ALL replaces NOT IN and in the absence of NULL has the same effect. It reads, somewhat ambiguously, as "not equal to all". In fact the expression yields TRUE if and only if the condition comparing StudentId values is TRUE for every row in the result of the subquery.

Effects of NULL

The treatment of NULL in invocations of EXCEPT is as for UNION. This is different from its treatment in those of NOT IN and quantified comparisons. In the case of *t1* EXCEPT *t2*, row *r* of *t1* appears in the result if and only if there does not exist a row *s* in *t2* such that *r* IS NOT DISTINCT FROM *s* evaluates to TRUE. Note that *x* NOT IN (*t*) evaluates to UNKNOWN whenever *x* = *x* does, including in particular the case where *x* IS NULL evaluates to TRUE because every field of the row *x* is the null value. Recall that when the condition given in a WHERE clause evaluates to UNKNOWN for row *r*, then *r* does not appear in the result.

Now, suppose that IS_ENROLLED_ON contains the rows ('S4', 'C1'), (NULL, 'C1'), and no other rows for course C1. Then 'S4' NOT IN (SELECT StudentId FROM IS_ENROLLED_ON WHERE CourseId = 'C1') evaluates to FALSE, but 'S4' <> ALL (SELECT StudentId FROM IS_ENROLLED_ON WHERE CourseId = 'C1') evaluates to UNKNOWN. So examples 4.11b and 4.11c are not equivalent in the presence of NULL. Nevertheless, Examples 4.11b and 4.11c are equivalent because WHERE treats FALSE and UNKNOWN alike—a WHERE condition applied to a row has to yield TRUE for the row to appear in the result.

Historical Notes

The grammar given in the appendix to the SEQUEL paper uses the mathematical symbol "-" as an alternative to ∪ for union, strongly suggesting that it stands for set difference. There is no mention of this operator in the body of the paper. However, EXCEPT was missing from original SQL and didn't appear in the standard until 1992. Curiously, EXCEPT without ALL is now a mandatory conformance feature while EXCEPT ALL and both varieties of INTERSECT (ALL and DISTINCT) remain optional ones. (INTERSECT is mentioned in the discussion on semijoin in Chapter 5, Section 5.2.) The membership tests using IN and NOT IN were in original SQL but not in SEQUEL. SEQUEL did, however, support quantified comparisons, those these were limited to rows and tables of degree one.

Download free eBooks at bookboon.com

4.11 Concluding Remarks

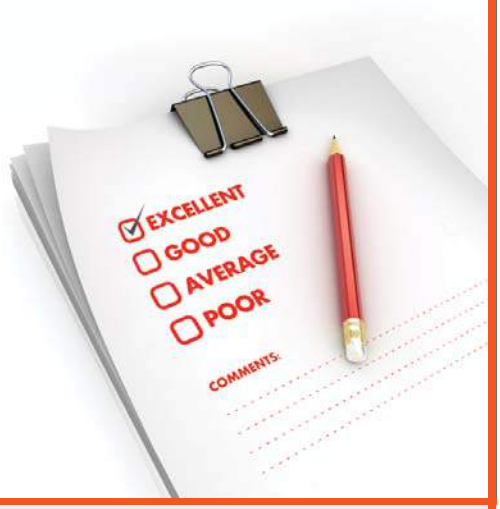
I have described how the following relational operators are supported, directly or indirectly, in SQL, noting various quirks in the language on the way.

- `JOIN` (via `NATURAL JOIN`)
- `RENAME` (via possibly laborious longhand)
- projection (via `SELECT DISTINCT`)
- restriction (`WHERE`)
- `EXTEND` (via possibly laborious longhand)
- `UNION` (via `UNION DISTINCT CORRESPONDNG`)
- semidifference (`NOT MATCHING` in **Tutorial D**—via `EXCEPT` for special cases, otherwise via possibly laborious longhand)

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now



Go to www.helpmyassignment.co.uk for more info

Helpmyassignment

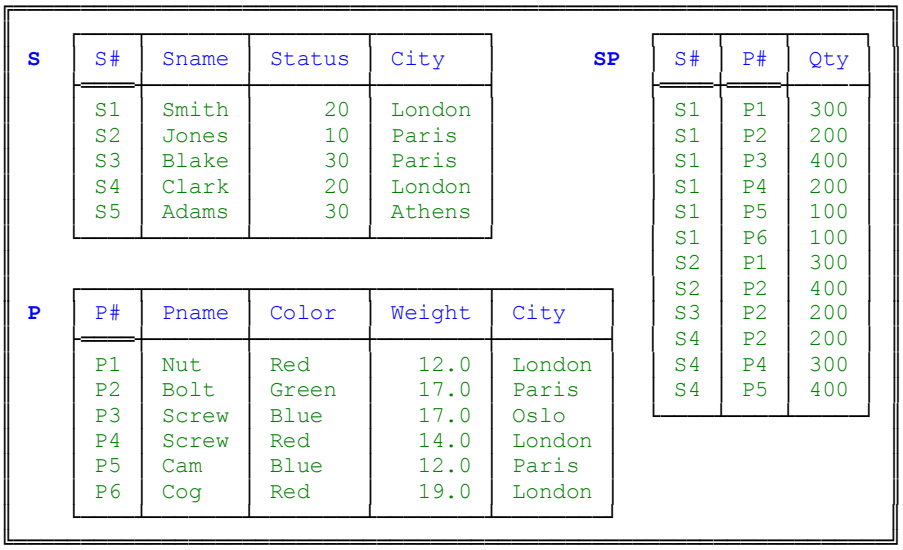
Download free eBooks at bookboon.com

Click on the ad to read more

The fact that SQL does support all of these operators, one way or another, makes SQL relationally complete, apart from its failure to support TABLE_DEE and TABLE_DUM, but I have noted several present-day features that were not in the early versions of the language, claiming that as a consequence those early versions were, unintentionally, not relationally complete. I have not yet fully explained that claim. That's because my explanation involves some of the material of the next chapter, where I look for SQL counterparts of those extra relational operators we include as "shorthands" in **Tutorial D**. See Section **5.6, Summarization in SQL**.

EXERCISES

1. Figure 4.13 shows the supplier-and-parts database from Chris Date's *Introduction to Database Systems (8th edition),* as shown on the inside back cover of that book (except that the attribute names there are in upper case). The exercises assume you have access to an SQL implementation.



S

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#	Qty
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

P#	Pname	Color	Weight	City
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Figure 4.13: The suppliers-and-parts database

Execute an SQL CREATE TABLE statement for each of S, P and SP. Use INTEGER as the declared type for STATUS and QTY, DECIMAL(5,2) for WEIGHT, and appropriate VARCHAR or CHAR types for all the other columns. Feel free to use lower case or mixed case to suit your own taste for column and table names, but do not otherwise change any of the given names.

Include primary key constraints as indicated by the underlining of column names in Figure 4.13.

"Populate" (as they say) each table with the values shown in Date's tables, using SQL INSERT statements.

2. Attempt to insert a row into SP with supplier number S1, part number P1 and quantity 100. Explain the result of your attempt.

Download free eBooks at bookboon.com

3. For each of the following **Tutorial D** expressions (taken from Exercise 5 in the theory book's exercises headed **Working with a Database in *Rel***), give an SQL expression that's as nearly equivalent as possible.

a) `SP WHERE P# = 'P2'`

b) `S { ALL BUT Status }`

c) `SP { S#, Qty }`

d) `P NOT MATCHING ( SP WHERE S# = 'S2' )`

e) `S MATCHING ( SP WHERE P# = 'P2' )`

f) `S { City } UNION P { City }`

g) `S { City } MINUS P { City }`

h)
```
( ( S RENAME { City AS SC } ) { SC } ) JOIN
( ( P RENAME { City AS PC } ) { PC } )
```

4. Write SQL expressions for the following queries. Compare your solutions with its counterpart in your solutions to Exercise 6 in the theory book's exercises headed **Working with a Database in *Rel***.

a) Get all shipments.

b) Get supplier numbers for suppliers who supply part P1.

c) Get suppliers with status in the range 15 to 25 inclusive.

d) Get part numbers for parts supplied by a supplier in Paris.

e) Get part numbers for parts not supplied by any supplier in Paris.

f) Get city names for cities in which at least two suppliers are located.

g) Get all pairs of part numbers such that some supplier supplies both of the indicated parts.

h) Get supplier numbers for suppliers with a status lower than that of supplier S1.

i) Get supplier-number/part-number pairs such that the indicated supplier does not supply the indicated part.

Download free eBooks at bookboon.com