5 Building on The Foundation

5.1 Introduction

Having described, in Chapter 4, operators that are both necessary and sufficient for relational completeness, the theory book builds on that foundation in Chapter 5 by describing some additional operators that have been found useful and are included in **Tutorial D**. These include some additional relational operators that are presented as *shorthands*—operators that can be defined in terms of ones already defined. For example, in Section 5.1, composition and semijoin are both defined in terms of the primitive operators, join and projection.

Here I show SQL counterparts for the **Tutorial D** expressions in Chapter 5, sometimes using longhands perforce. As we shall see, quite a few of the SQL examples use operators that didn't exist in SQL until 1999 or later. Moreover, these late additions are not shorthands—before they arrived, some of the examples in Chapter 5 of the theory book could not be expressed at all in SQL.

In the theory book at this point we needed a couple of additional relvars in our example database, `COURSE` and `EXAM_MARK`, along with the existing `IS_CALLED` and `IS_ENROLLED_ON`. They are shown again here, now as SQL tables, in Figure 5.1.

COURSE

CourseId	Title
C1	Database
C2	HCI
C3	Op systems
C4	Programming

EXAM_MARK

StudentId	CourseId	Mark
S1	C1	85
S1	C2	49
S1	C3	85
S2	C1	49
S3	C3	66
S4	C1	93

Figure 5.1: Current values of base tables COURSE and EXAM_MARK

The predicate for `COURSE` is "Course *CourseId* is entitled *Title*." The predicate for `EXAM_MARK` is "Student *StudentId* sat the exam for course *CourseId* and scored *Mark* marks for that exam." The SQL definitions for these tables are

```
CREATE TABLE COURSE ( CourseId CID,
                      Title VARCHAR(100) NOT NULL,
                      PRIMARY KEY ( CourseId )
                    ) ;

CREATE TABLE EXAM_MARK ( StudentId SID,
                         CourseId CID,
                         Mark INTEGER NOT NULL,
                         PRIMARY KEY ( StudentId, CourseId )
                       ) ;
```

(`SID` and `CID` are now assumed to be SQL domain names, defined on type `VARCHAR(5)`. As not many SQL implementations support domains, in practice you would be more likely to see some `CHAR` type being used here. Also, the table definitions might include some additional constraint definitions, but those are dealt with in Chapter 6.)

5.2 Semijoin and Composition

For semijoin **Tutorial D** has the dyadic relational operator `MATCHING`, defined thus:

> *r1* **`MATCHING`** *r2,* where *r1* and *r2* are relations such that *r1* `JOIN` *r2* is defined, is equivalent to
> (*r1* **`JOIN`** *r2*) { *r1-attrs* }
> where *r1-attrs* is a commalist containing all and only the attribute names of *r1*.

and the example

```
COURSE MATCHING EXAM_MARK
```

is given as a relational expression for the predicate, "There exist a student *StudentId* and a mark *Mark* such that *StudentId* sat the exam and scored *Mark* marks for course *CourseId* and *CourseId* is entitled *Title*" (which could be abbreviated to "At least one student sat the exam for Course *CourseId*, entitled *Title*"). The resulting relation consists of just those tuples in COURSE that have at least one matching tuple in EXAM_MARK.

In SQL semijoins usually have to be done in longhand and there are several ways of doing them. The simplest way is to use the comparison operator IN, SQL's counterpart of **Tutorial D**'s ∈ (which *Rel* implements as IN):

```
SELECT * FROM COURSE
WHERE CourseId IN ( SELECT CourseId From EXAM_MARK )
```

or, equivalently, the quantified comparison operator, =ANY:

```
SELECT * FROM COURSE
WHERE CourseId =ANY ( SELECT CourseId From EXAM_MARK )
```

The result in either case is the table shown in Figure 5.2. Note in passing that the table operand of IN or = ANY here contains three appearances of the row for course C1, and thus does not represent a relation. That could be fixed by specifying DISTINCT, of course, but here the redundant duplicates are obviously not a problem. Note the need to write the column name CourseId twice. Perhaps this burden is compensated for to some extent by the improved clarity.

CourseId	Title
C1	Database
C2	HCI
C3	Op systems

Figure 5.2: Semijoin of COURSE with EXAM_MARK

Translating the SQL longhand back into **Tutorial D**, we get

```
COURSE WHERE TUPLE{CourseId CourseId} ∈ EXAM_MARK {CourseId}
```

In spite of appearances, `TUPLE{CourseId CourseId}` is a reasonable translation of SQL's plain `CourseId` because in standard SQL the first operand of `IN` is in fact a row. A row is denoted in general by a commalist of expressions enclosed in parentheses, optionally preceded by the key word `ROW`, but the parentheses can be omitted when the row consists of just one field, as explained in Chapter 2, Section **2.9 Table Literals**. However, many SQL implementations fail to support rows with more than one field as the first operand of `IN`, in which case a longer longhand is needed. For example, to obtain enrolments for which an exam mark is available (`IS_ENROLLED_ON MATCHING EXAM_MARK` in **Tutorial D**), we might expect to be able to write

```
SELECT * FROM IS_ENROLLED_ON
WHERE (CourseId, StudentId) IN
      ( SELECT CourseId, StudentId From EXAM_MARK )
```

but instead we are forced to write, in such an implementation,

```
SELECT * FROM IS_ENROLLED_ON A
WHERE EXISTS ( SELECT * FROM EXAM_MARK B
               WHERE A.CourseId = B.CourseId
                 AND A.StudentId = B.StudentId )
```

Also, in the formulation using IN, note carefully the need to pay attention to order in which column names are written—the correspondence between the fields of the row and the columns of the table is by position, not name, and in any case the fields of the row do not necessarily inherit the names CourseId and StudentId, as we shall see in many more examples in this chapter.

Alternatively, by translating the definition given for semijoin into SQL, we could write

```
SELECT DISTINCT A.*
FROM   IS_ENROLLED_ON AS A NATURAL JOIN EXAM_MARK AS B
```

and now the need to mention column names has gone away.

Finally, in the special case where all the columns of the first operand are common columns, then we can use SQL's INTERSECT operator. As this is indeed the case with the example at hand, we can write

```
SELECT * FROM IS_ENROLLED_ON
INTERSECT CORRESPONDING
SELECT StudentId, CourseId FROM EXAM_MARK
```

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

t1 INTERSECT DISTINCT *t2* returns the table consisting of a single appearance of each row that appears in both *t1* and *t2*. *t1* INTERSECT ALL *t2* returns the table consisting of *m* appearances of each row *r* that appears in both *t1* and *t2*, where *m* is the smaller of its number of appearances in *t1* and its number of appearances in *t2*.

Historical Notes

The SEQUEL paper uses the mathematical symbol "∩" in place of SQL's INTERSECT. However, like EXCEPT, INTERSECT was missing from original SQL and didn't appear in the standard until 1992. It remains an optional conformance feature. The IN operator was in original SQL but had no counterpart in SEQUEL. Standard SQL allows (CourseId, StudentId) =ANY in place of (CourseId, StudentId) IN and defines the two formulations to be equivalent. SEQUEL's support for such "quantified comparisons" was limited to rows and tables of degree one. Also, SEQUEL did not use a key word such as ANY (standard SQL admits ANY or SOME, synonymously) but rather assumed existential quantification when the second operand of a comparison operator was a table and ALL was not specified.

Turning now to the operator known as composition, the theory book gives the example

```
COURSE COMPOSE EXAM_MARK
```

as representing the predicate "Student *StudentId* scored *Mark* marks in the exam for a course entitled *Title*." The corresponding relation must have attributes `StudentId`, `Mark`, and `Title`. The first two would clearly be derived from `EXAM_MARK`, the third from `COURSE`.

The result is shown in Figure 5.3. As you can see, it is equivalent to the join of `COURSE` and `EXAM_MARK`, projected over all but the common attribute, `CourseId`.

Title	StudentId	Mark
Database	S1	85
HCI	S1	49
Op systems	S1	85
Database	S2	49
Op Systems	S3	66
Database	S4	93

Figure 5.3: Composition of COURSE and EXAM_MARK

In SQL, as the result contains data from both operand tables, this time we really do have to specify both tables in the `FROM` clause—neither of the operators `IN` and `EXISTS` is of any use here.

```
SELECT DISTINCT Title, StudentId, Mark
FROM   IS_ENROLLED_ON NATURAL JOIN EXAM_MARK
```

Now, it is worth repeating here the theory book's justification for including `COMPOSE` in **Tutorial D.**

In case you are wondering if `COMPOSE` really is useful enough to be worth including in a computer language, and therefore to be worthy of inclusion in textbooks like this one, an important part of the motivation for its inclusion in **Tutorial D** was a desire to illustrate the *extensibility* of a well-designed language. Adding new operators increases a language's complexity, to be sure, but that added complexity can be compensated for if the new operators are not only useful but can be easily defined and taught in terms of what the user already knows.

We can note in passing that it would be easy to extend SQL to support semijoin and composition explicitly—just allow words such as `MATCHING` and `COMPOSE` to appear where `NATURAL JOIN` is currently permitted!

Effects of `NULL`

Observations similar to those given in connection with `UNION` and `EXCEPT` in Chapter 4, Sections 4.9 and 4.10, apply here too. For example, consider the following two expressions:

```
SELECT x FROM T1
INTERSECT
SELECT x FROM T2

SELECT x FROM T1
WHERE x IN
( SELECT x FROM T2 )
```

If row *r* appears in both `T1` and `T2` and satisfies the condition `x IS NULL`, then *r* appears in the result in the first case, using `INTERSECT`, but not in the second, because the condition in the `WHERE` clause evaluates to `UNKNOWN` for that row.

5.3 Aggregate Operators

SQL supports all of the aggregate operators mentioned in the theory book and many more besides. The syntax, however, involves an unusual trick that SQL calls a *scalar subquery.* A scalar subquery is a table expression that satisfies all of the following conditions:

- It is enclosed in parentheses.
- It appears where a scalar expression is expected.
- The result of the enclosed table expression has exactly one column and at most one row.

The result of a scalar subquery is then either the single column value appearing in that single row or, when there is no row, `NULL`. Examples 5.1 to 5.4 are the SQL counterparts of those examples in the theory book. In each case the required aggregation is specified in the `SELECT` clause of a scalar subquery and the table operand is specified in the clause(s) following that `SELECT` clause.

Example 5.1: Counting the rows in EXAM_MARK

```
( SELECT COUNT(*) FROM EXAM_MARK )
```

The table expression inside the parentheses in Example 5.1 is analogous to an invocation of `SUMMARIZE` in **Tutorial D** that specifies `BY { }`, as in

```
SUMMARIZE EXAM_MARK BY { } : { X := COUNT( ) }
```

and in fact standard SQL allows `GROUP BY ( )` to be added immediately before the closing parenthesis:

```
( SELECT COUNT(*) FROM EXAM_MARK GROUP BY ( ) )
```

I could have added `AS X` to the `SELECT` clause, to make the analogy exact, but that would have been pointless because of course the column name disappears in the process of converting the table to a number. Similar comments apply to Examples 5.2, 5.3, and 5.4. (SQL's `GROUP BY` construct is described in Section 5.6.)

Remember that the SQL Example 5.1 denotes a number, as opposed to a table, only when the context is appropriate—for example, when it appears on the right-hand side of an assignment to an integer variable, or as an operand in a comparison of two integers. When it appears as an element of a `FROM` clause, for example, or as an operand of `UNION`, then the enclosing parentheses do not affect the semantics and it denotes a table with one unnamed column.

`COUNT(*)` denotes the cardinality of the operand table. If the * is replaced by some expression *x*, then `COUNT(`*x*`)` denotes the number of rows that satisfy the condition *x* `IS NOT NULL`. For obvious reasons, that particular aspect of aggregation in SQL has no counterpart in **Tutorial D**.

Example 5.2: Counting the students who have scored more than 50 in some exam

```
( SELECT COUNT(*) FROM
    ( SELECT DISTINCT StudentId
      FROM   EXAM_MARK
      WHERE  Mark > 50 ) AS T )
```

Example 5.2 is a direct translation of the corresponding example in the theory book but it can be abbreviated—and is much more likely to be written—as

```
( SELECT COUNT ( DISTINCT StudentId )
  FROM   EXAM_MARK
  WHERE  Mark > 50 )
```

Historical Notes

The abbreviated formulation for Example 5.2 has been in SQL from the start. The longer form became available in SQL:1992, when support for "derived tables in the `FROM` clause" was introduced. Note that the longer form is needed when `SELECT DISTINCT` specifies more than one expression.

The use of parentheses in the `GROUP BY` clause, and support for `GROUP BY ( )` in particular, were introduced in SQL:1999. Previously, a `GROUP BY` clause had been required to specify at least one column.

Example 5.3 illustrates the use of `SUM` in SQL

Example 5.3: Adding up all the marks obtained by student S1

```
( SELECT SUM(Mark) FROM EXAM_MARK WHERE StudentId = 'S1' )
```

Effect of NULL

As with `COUNT`, the evaluation of `SUM(`*x*`)` ignores rows that satisfy the condition *x* `IS NULL`, this being a general rule applying to all aggregations in SQL. However, this doesn't mean that `SUM(`*x*`)` is guaranteed never to result in `NULL`, for `NULL` is the result whenever the operand table is empty or *x* `IS NULL` is *true* for every row. Thus, Example 5.3 results in `NULL` whenever student S1 has taken no exams.

Note that `SUM(`*x*`) + SUM(`*y*`)` is not in general equivalent to `SUM(`*x* + *y*`)`, because *x* + *y* evaluates to `NULL` when either *x* evaluates to `NULL` or *y* does. Thus, some values that are included in the summing of *x* and the summing of *y* might be omitted in the summing of *x* + *y*.

Example 5.4: MAX and MIN

```
( SELECT MAX(Mark) FROM EXAM_MARK WHERE StudentId = 'S1' )
( SELECT MIN(Mark) FROM EXAM_MARK WHERE StudentId = 'S1' )
```

Example 5.4 needs no further explanation. SQL also has `AVG` for averages. Its counterparts of **Tutorial D**'s aggregate `AND` and `OR` are spelled, respectively, `EVERY` and either `SOME` or `ANY`, but all of these must be used with care because of the consequences of the aforementioned general rule concerning the treatment of `NULL`. For example, if the condition *`c`* evaluates to `UNKNOWN` for every row of table *`t`*, or *`t`* is empty, then `( SELECT EVERY(`*`c`*`) FROM `*`t`*` )` evaluates to `UNKNOWN`, whereas when *`t`* is empty it really ought to evaluate to `TRUE`. (I hesitate to hazard a guess as to what it should "really" evaluate to when *`t`* is nonempty but *c* evaluates to `UNKNOWN` in every row.)

At this point the theory book gives Example 5.5 (not repeated here) as an unpleasant way of computing the number of students who sat each exam and shows Figure 5.4, repeated here, as a preferred way of presenting the result of such a computation.

CourseId	n
C1	3
C2	1
C3	1
C4	0

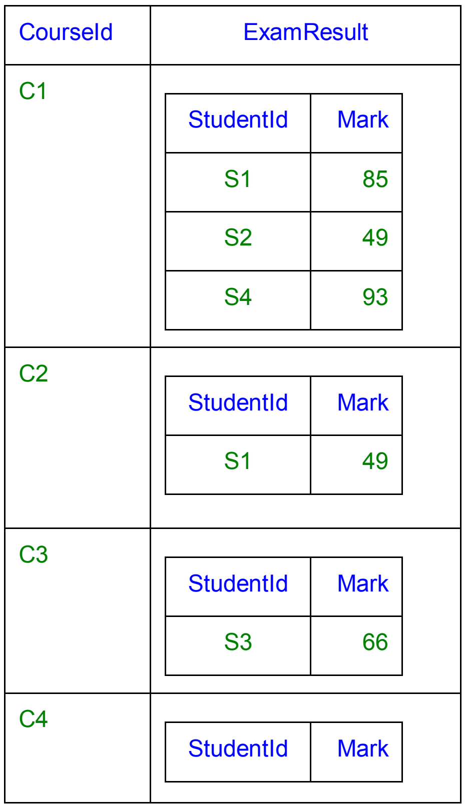
Figure 5.4: How many sat each exam

The example is used as motivation for **Tutorial D**'s `SUMMARIZE` operator but the demonstration that `SUMMARIZE` is indeed a shorthand is rather elaborate, occupying the next three sections of the theory book. The first of those three, Section 5.4, **Relations within a Relation**, describes attributes whose values are relations and shows how such attributes can be obtained using relational extension. Section 5.5, **Using Aggregate Operators with Nested Relations**, then shows how results such as that shown in Figure 5.4 can be obtained using relational operators described in Chapter 4, and then `SUMMARIZE` is introduced in Section 5.6 as a shorthand for obtaining those same results.

As we shall eventually see, SQL has a fairly direct counterpart of `SUMMARIZE BY`, using aggregation in combination with a `GROUP BY` clause. SQL textbooks do not normally teach aggregation and `GROUP BY` along the theory book's lines for teaching `SUMMARIZE`, but I do so here to maintain the parallel structure and also to show how the intermediate results of sections 5.4 and 5.5 can be obtained in modern SQL. The names of these sections are SQL paraphrases of their counterparts in the theory book.

5.4 Tables within a Table

Figure 5.5 here is an exact copy of the one in the theory book and as before it is just an alternative way of representing some of the information conveyed by the tables in Figure 5.1 (but not a recommended database design for that information).



CourseId	ExamResult
C1	StudentId | Mark S1 | 85 S2 | 49 S4 | 93
C2	StudentId | Mark S1 | 49
C3	StudentId | Mark S3 | 66
C4	StudentId | Mark

Figure 5.5: Tables within a table

For each course, it shows the exam mark obtained by each student who took the exam for that course. Example 5.6 shows how to obtain this table—let's call it C_ER again—in SQL.

Example 5.6: Obtaining C_ER from COURSE and EXAM_MARK

```
SELECT  CourseId,
        CAST (
          TABLE ( SELECT  DISTINCT  StudentId,  Mark
                  FROM    EXAM_MARK  AS  EM
                  WHERE   EM.CourseId  =  C.CourseId )
          AS  ROW  ( StudentId  SID,  Mark  INTEGER  )  MULTISET  )
        AS  ExamResult
FROM    COURSE  AS  C
```

Explanation 5.6

- The `SELECT` clause operates on each row of the result of the `FROM` clause—i.e., on each row of the `COURSE` table, deriving two columns, `CourseId` and `ExamResult`.
- **`CourseId`** is self-explanatory, merely carrying forward the column values from the column of that name in `COURSE`.
- **`TABLE ( SELECT DISTINCT StudentId, Mark FROM EXAM_MARK AS EM WHERE EM.CourseId = C.CourseId )`** denotes a multiset whose elements are rows, obtained by taking the `StudentId` and `Mark` values from those rows of `EXAM_MARK` that match the current row of `COURSE` on `CourseId`. **Note very carefully,** however, that this multiset does not necessarily inherit the column names, `StudentId` and `Mark`, from the table that is the operand to the invocation of `TABLE`. The SQL standard allows the column names to be "implementation-dependent" (i.e., undefined) so long as no two columns have the same name. An implementation that nevertheless carried forward the unique names `StudentId` and `Mark` would be both sensible and conforming, and would obviate the need for the `CAST` invocation explained in the next bullet.

  The same multiset would result if the word `DISTINCT` had been omitted, thanks to the `WHERE` condition, but I include it because the example in the theory book uses `COMPOSE`, which is defined as a projection of a join, and SQL's counterpart of projection uses `SELECT DISTINCT`.
- **`CAST ( `*`t`*` AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )`**, where *t* is the above `TABLE` expression, addresses the aforementioned possible problem by assigning the required column names. Note that we need to know and write down the declared types of those columns as well as their names. The "type conversion" operator `CAST` is described in Chapter 2, Explanation 2.1a. Here it is being used to convert a value of some incompletely defined multiset type to one whose multiset type is explicitly defined.
- **`AS ExamResult`** then gives the resulting column the name `ExamResult`. Note that here the name comes after `AS` and the expression defining it comes before, in the same style as the use of `AS` to define the range variables `C` and `EM` in the example.

The values for columns such as `ExamResult` in this example have sometimes been referred to informally as *nested tables,* being "tables within a table", so to speak. Unfortunately, however, they are not actually tables, but rather multisets of rows. Because of that fact, a column such as `ExamResult` cannot appear as an element in a `FROM` clause, so we cannot use it in the way it is used in Example 5.7 in the theory book.

Historical Notes

The TABLE operator didn't appear in SQL until SQL:2003 and it remains an optional conformance feature. See Chapter 2, Section **2.6 What Is a Type?** and Section **2.8, The Type of a Table.**

It is not surprising that support for nested tables was absent from the early SQL implementations. Codd's *first normal form* (1NF) had been widely understood to be a required property of all relations, not just database relvars, and relation-valued attributes were proscribed under that normal form. The proscription came into question during the 1980s, when the term "not first normal form" was coined, abbreviated to NFNF and thence, somewhat jocularly, to NF^2.

CAST first appeared in SQL:1992.

5.5 Using Aggregation on Nested Tables

Example 5.7 is the most direct translation of its counterpart in the theory book that can be obtained in SQL but it is so over-elaborate that no SQL practitioner would consider using it. It uses the aggregate operator COUNT on the table values for column ExamResult to obtain the number of students who sat each exam. Unfortunately, as already noted, we cannot operate directly on ExamResult as a FROM clause element. Instead, we need to use an artifice that is specially devised for the sake of this example.

Example 5.7: How many students sat each exam (not a recommended solution!)

```
WITH C_ER AS (
  SELECT CourseId,
         CAST (
          TABLE ( SELECT DISTINCT StudentId, Mark
                  FROM   EXAM_MARK AS EM
                  WHERE  EM.CourseId = C.CourseId )
          AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )
        AS ExamResult
  FROM   COURSE AS C )
SELECT CourseId, (SELECT COUNT(*)
                 FROM   TABLE(ER(ExamResult)) AS t) AS n
FROM   C_ER
```

Explanation 5.7

- The `WITH` clause, occupying the first nine lines of the example, illustrates SQL's counterpart of **Tutorial D**'s construct of the same name. It assigns the name, `C_ER`, to the result of Example 5.6. That name, `C_ER`, is then used in the `FROM` clause of the expression that follows the `WITH` clause. Note that here the name comes before `AS` and the expression defining it comes after. This is consistent with the analogous use of `AS` in `CREATE VIEW` statements, SQL's counterparts of **Tutorial D**'s virtual relvar definitions.

- **`TABLE(ER(ExamResult))`** seems to be the only way of having a multiset valued column operated on as an element of a `FROM` clause—a simple column name is not allowed to appear here. `TABLE(ExamResult)` can't be used either, because when an invocation of `TABLE` appears as a `FROM` clause element, its operand is required to be, specifically, an invocation of a user-defined function. Here I am assuming that `ER` is defined as follows:

```
CREATE FUNCTION ER
( SM ROW ( StudentId SID, Mark INTEGER ) MULTISET )
RETURNS TABLE ( StudentId SID, Mark INTEGER )
RETURN SM ;
```

The type name `TABLE ( StudentId SID, Mark INTEGER )` is actually just a synonym for `ROW ( StudentId SID, Mark INTEGER ) MULTISET )`. The misleading synonym is available only in a `RETURNS` clause and not as a parameter type, for example. So `ER` is actually a no-op, returning its input.

- **(SELECT COUNT(*) FROM TABLE(ER(ExamResult)) AS t)** is a scalar subquery, yielding the cardinality of the multiset of rows that is the value of the column ExamResult in the current row of C_ER. Because we are using the expression to denote a scalar value rather than a table, naming the column would be pointless (apart, perhaps, from injecting a somewhat sarcastic element of purism). As COUNT(*) doesn't use a column name, Example 5.7 is valid even if we omit the invocation of CAST to assign column names.
- **AS n** then gives the resulting column the name n. Note that here the name comes after AS and the expression defining it comes before, in the same style as the use of AS to define the range variables C and EM in the example.

So long as CAST is used as shown, we could obtain the total marks for each exam in similar fashion, using SUM(Mark) AS TotalMarks. However, this gives NULL, instead of zero, for the courses whose exams nobody sat. That problem can be addressed by using COALESCE, as shown in Example 5.7a.

Example 5.7a: Give the total of marks for each exam (still not a recommended solution)

```
WITH C_ER AS (
  SELECT CourseId,
         CAST (
           TABLE ( SELECT DISTINCT StudentId, Mark
                   FROM   EXAM_MARK AS EM
                   WHERE  EM.CourseId = C.CourseId )
           AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )
         AS ExamResult
  FROM   COURSE AS C )
SELECT CourseId, COALESCE (
                      (SELECT SUM(Mark)
                       FROM   TABLE(ER(ExamResult)) AS t), 0)
AS n
FROM   C_ER
```

Explanation 5.7a

- **COALESCE((SELECT SUM(Mark) FROM TABLE(ER(ExamResult)) AS t),0)** yields the value of the scalar subquery whenever ExamResult is nonempty, otherwise zero. SQL's COALESCE is an *n*-adic operator that takes a commalist of expressions of the same declared type and yields the result of the first of those expressions, in the order in which they are written, that does not evaluate to NULL. When there is no such operand, it yields NULL anyway, of course.

Notice that SQL does not follow the rule given in the theory book whereby aggregation over the empty relation yields the identity value for the basis operator (when such a value exists), in this case addition, whose identity value is zero.

Now, as I remarked already, examples 5.7 and 5.7a are not recommended ways to obtain the required result. Example 5.7b shows how 5.7a can be condensed into a much more concise solution.

Example 5.7b: Give the total of marks for each exam (simplified solution)

```
SELECT  CourseId,
        COALESCE ( ( SELECT SUM(Mark)
                     FROM   EXAM_MARK AS EM
                     WHERE  EM.CourseId = C.CourseId ),
                   0 ) AS TotalMark
FROM    COURSE AS C
```

Explanation 5.7b

- The first operand to the invocation of `COALESCE` is a scalar subquery similar to those shown in Section **5.3, Aggregate Operators**. Recall the need to enclose the `SELECT` expression in parentheses, without which it would denote a table rather than a number.

Historical Notes

The SQL standard's `WITH` clause first appeared in SQL:1999. It remains an optional conformance feature. Moreover, even if an implementation does support it, it is further optional as to whether it is permitted to appear in a subquery.

Scalar subqueries have been in SQL from the beginning but were originally allowed to appear only as the second operand of a comparison, and thus could not appear in the `SELECT` clause. Moreover, they were subject to various arbitrary restrictions, such as not being allowed to contain `UNION`, `GROUP BY`, or `HAVING`. The restrictions were lifted in SQL:1992 but implementations that retained them were still able to claim the minimum level of conformance until the appearance of SQL:1999.

The `COALESCE` operator was added to the language in SQL:1992. It ceased to be optional in SQL:1999.

If we need the average mark for each exam we will have to avoid zero-divides by excluding those which no students sat. As in the theory book we can do that by homing in on just those `CourseId` values that appear in `EXAM_MARK`, as shown in Example 5.8, the differences from Example 5.7b being shown in bold.

Example 5.8: Average mark per exam

```
SELECT CourseId,
       ( SELECT AVG(Mark)
           FROM   EXAM_MARK AS EM
           WHERE  EM.CourseId = C.CourseId ) AS AvgMark
FROM   EXAM_MARK AS C
```

Now, the theory book goes on from here to describe two varieties of the relational summarization operator—SUMMARIZE PER and SUMMARIZE BY in **Tutorial D**—providing useful shorthands for expressions like Example 5.7. SQL has no such operators but it does provide a useful shorthand for cases that in **Tutorial D** can be formulated using SUMMARIZE BY, as we shall see in the next section. As with SUMMARIZE, this shorthand also allows multiple aggregations to be specified on the same table without repeating the expression denoting that table. Example 5.8a shows how this repetition problem arises if we followed the style of Example 5.8 to obtain both the average mark and the total, this time ignoring exams that nobody sat.

Example 5.8a: Average and total mark per exam (not a recommended solution!)

```
SELECT CourseId,
       ( SELECT AVG(Mark)
           FROM   EXAM_MARK AS EM
           WHERE  EM.CourseId = C.CourseId ) AS AvgMark,
       ( SELECT SUM(Mark)
           FROM   EXAM_MARK AS EM
           WHERE  EM.CourseId = C.CourseId ) AS TotalMark
FROM   EXAM_MARK AS C
```

5.6 Summarization in SQL

Example 5.9 in the theory book uses **Tutorial D**'s SUMMARIZE PER to give the same result as Example 5.7. Because SQL has no direct counterpart of SUMMARIZE PER, here I need to go straight to Example 5.11 to show how SUMMARIZE BY invocations can be simulated in SQL. Then I can show how Example 5.9 could be translated in SQL. Example 5.11 introduces us to SQL's GROUP BY clause, this being its direct counterpart of **Tutorial D**'s BY specification in SUMMARIZE BY.

Example 5.11: Average mark for each exam, using GROUP BY (recommended solution)

```
SELECT    CourseId, AVG(mark) AS AvgMark
FROM      EXAM_MARK
GROUP BY  CourseId
```

The `GROUP BY` clause is not to be confused with **Tutorial D**'s `GROUP` operator. Actually, `GROUP BY CourseId` can be considered to have the same effect as `GROUP{ALL BUT CourseId}` in **Tutorial D**, in which case `AVG(Mark)` then operates on the nested tables produced by the `GROUP BY` clause. However, the SQL standard and most SQL textbooks do not define `GROUP BY` in such terms. Rather, they introduce the notion of partitioning the body of a table into *groups* and refer to such a partitioned table as a *grouped table*. When a `SELECT` expression includes a `GROUP BY` clause, each `SELECT` clause element must specify a column that is functionally dependent on the `GROUP BY` columns, thus reducing each group to a single row. (Functional dependence is taught in Chapter 7 of the theory book. It should be clear to you that the FD {`CourseId`} → { `CourseId, AvgMark` } always holds in the result of Example 5.11.)

We can return to the theory book's Example 5.9 now because we can use `GROUP BY` to obtain part of the required result and an outer join (see Chapter 4, Example 4.1e) in conjunction with `COALESCE` to complete it. The example as given could be addressed in similar fashion to Example 5.7 but the method shown in Example 5.9 is likely to be preferred when more than one aggregation is to be specified on the same table.

Example 5.9: How many students sat each exam,
using GROUP BY, NATURAL LEFT JOIN, and COALESCE

```
SELECT CourseId, COALESCE(n, 0) AS n
FROM   COURSE NATURAL LEFT JOIN
      ( SELECT    CourseId, COUNT(*) AS n
        FROM      EXAM_MARK
        GROUP BY  CourseId ) AS T
```

Explanation 5.9

- **`NATURAL JOIN`** is described in Chapter 4, Section 4.1. Note, however, that the use of `LEFT` makes this an outer join, whereas Codd's term *natural join* referred to the "inner" variety only.

- **`LEFT`** specifies that each unmatched row in the first join operand, `COURSE`, is to be extended with `NULL` for the column `n`.

- **`COALESCE (n, 0) AS n`** effectively replaces those appearances of `NULL` by the correct value, 0.

(I give no counterpart for Example 5.10 in the theory book because it merely shows how **Tutorial D**'s `SUMMARIZE BY` is just a shorthand for certain special cases of `SUMMARIZE PER`.)

Historical Notes

In the final section of Chapter 4, **Concluding Remarks,** I mentioned a claim that "early versions" of SQL were not relationally complete, for reasons beyond the "small" matter of failing to recognize the existence of relations of degree zero (or tables with no columns). I can now give the rationale for that claim. Example 5.9 uses a SELECT expression in its FROM clause, which was not supported by the international standard until SQL:1992 appeared. Without such support it is not always possible for the table resulting from such an expression to be joined with another table. In particular, it is not possible when a join is required of two tables that are both obtained by use of SELECT … FROM … GROUP BY.

5.7 Grouping and Ungrouping in SQL

Example 5.6a is derived from Example 5.6 by specifying EXAM_MARK in place of COURSE in the main FROM clause.

Example 5.6a: Obtaining C_ER2 from EXAM_MARK

```
SELECT  CourseId,
        CAST (
         TABLE ( SELECT DISTINCT StudentId, Mark
                 FROM   EXAM_MARK AS EM2
                 WHERE  EM1.CourseId = EM2.CourseId )
         AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )
        AS ExamResult
FROM    EXAM_MARK AS EM
```

Figure 5.4 shows the result, named C_ER2 for convenience. It differs from the C_ER of Figure 5.3 only in the absence of a row for course C4, whose exam nobody sat.

CourseId	ExamResult	
C1	StudentId	Mark
	S1	85
	S2	49
	S4	93
C2	StudentId	Mark
	S1	49
C3	StudentId	Mark
	S3	66

Figure 5.4: Intermediate result C_ER2 from Example 5.6a

The theory book gives Example 5.12 to illustrate use of the operator GROUP as a shorthand for obtaining such results. SQL has no direct counterpart of GROUP but Example 5.12 shows how to use GROUP BY to obtain an alternative formulation to produce C_ER2 that is more concise than Example 5.8a.

Example 5.12: Using GROUP BY and COLLECT to obtain C_ER2

```
SELECT    CourseId,
          CAST (
             COLLECT(ROW(StudentId, Mark))
             AS ROW ( StudentId SID, Mark INTEGER ) MULTISET )
          AS ExamResult
FROM      EXAM_MARK
GROUP BY  CourseId
```

Explanation 5.12

- **ROW(StudentId, Mark)** forms the row consisting of the StudentId and Mark values of the current row of EXAM_MARK, in that order. The two fields of this row are unnamed.
- **COLLECT(ROW(StudentId, Mark))** collects together as a multiset all of those rows that are derived EXAM_MARK rows having the same CourseId value. In fact it is shorthand for FUSION(ROW(StudentId, Mark) MULTISET), where FUSION is SQL's nearest counterpart of **Tutorial D**'s aggregate UNION. For each value of its operand, COLLECT derives the multiset containing just that value, and returns the FUSION (see next bullet) of all the multisets thus formed.
- FUSION is aggregate multiset union (UNION ALL), not UNION per se. In general the same value (in our example, a row) might appear more than once in the result of a COLLECT invocation. Fortunately, that won't happen here because the same StudentId, Mark combination cannot appear along with the same CourseId in more than one row of EXAM_MARK, so DISTINCT could be omitted.

  Tutorial D's aggregate operator UNION is not mentioned in the theory book. It is used for taking the union of the relations appearing as values of a specified attribute in the relation operand of that aggregate operator.
- **CAST (*m* AS ROW (StudentId SID, Mark INTEGER) MULTISET)**, where m is the above COLLECT expression, names the columns of the nested table, ExamResult. Note the need to spell out the entire declared type of ExamResult, even though it differs from that of the COLLECT expression only in the names of the two columns.

By the way, the following table expression:

```
SELECT *
FROM ( FUSION ( TABLE ( VALUES ( StudentId, Mark ) ) ) )
     AS T(StudentId, Mark)
```

denotes the same value as the given CAST expression but unfortunately it cannot be used as an alternative for the very reason that it is a table expression—table expressions are not permitted as SELECT clause elements. We could enclose it in parentheses following the word TABLE, but then, as I have already explained, we have no guarantee that the column names would be propagated to the result and so we might have to use CAST again after all!

That table expression is rather convoluted. You might prefer not to be given an explanation for it but here it is anyway:

`( StudentId, Mark )` denotes the row consisting of the `StudentId` value followed by the `Mark` value. Putting VALUES in front of it makes it into table expression denoting the table containing just that row. Putting TABLE in front of that makes it into a multiset expression, as required by FUSION. Although table expressions are not permitted as SELECT clause elements, multiset expressions *are* permitted as FROM clause elements, allowing us to enclose the FUSION invocation in `SELECT * FROM ( … ) AS T(StudentId, Mark)` to make sure we have the required column names.

In **Tutorial D**, the inverse operator of GROUP is UNGROUP. SQL has an operator, UNNEST, that can be used for similar purposes, but its method of invocation is somewhat peculiar, as Example 5.13 shows, and it can be used only to specify a FROM clause element.

Example 5.13: Inverse of Example 5.12, using UNNEST

```
SELECT DISTINCT * FROM C_ER2, UNNEST ( ExamResult ) AS M
```

The name C_ER2 could be defined using a WITH clause, as in Example 5.7a. Notice how the second element of the FROM clause has to be reevaluated for each row of C_ER2, whereas each FROM clause element is normally evaluated just once because its value does not vary from row to row of previous elements. The column reference `ExamResult` is a reference to the column of that name in C_ER2 and is permitted only because C_ER2 is specified *before* `UNNEST ( ExamResult )` in the FROM clause—a switching of these two FROM clause elements would result in a syntax error.

Effect of `NULL`

In Example 5.13, ExamResult is a column of type `ROW ( StudentId SID, Mark INTEGER ) MULTISET )`. In `C_ER2` `NULL` cannot appear in place of a value for that column, but in general `NULL` can appear in place of a value for a column of some multiset type. So we need to know what happens when `NULL` is given as the argument to an invocation of `UNNEST`. At the time of writing (in 2012), the SQL standard appears to be silent on that issue. One would expect it to give rise to an exception condition.

Note also that `NULL` might in general appear as an element of a multiset whose element type is a `ROW` type, though again this cannot arise in `C_ER2`, assuming that `C_ER2` is derived as shown in Example 5.12.

Historical Note

`UNNEST` first appeared in SQL:1999, though in that edition it was used only with arrays, not with multisets. `COLLECT` and `FUSION` first appeared in SQL:2003, along with `INTERSECTION`, for computing an aggregate intersection of multisets (and not to be confused with the table operator `INTERSECT`). They are all optional conformance features.

Now, the theory book at this point observes that the cardinality of the result of Example 5.13 is equal to the sum of the cardinalities of the `ExamResult` values in `C_ER2`. It then poses the following question as an *exercise for the reader:* Is it *always* the case that the cardinality of an ungrouping is equal to the sum of the cardinalities of the relations the operand relation is being ungrouped on? Although I didn't need to include `DISTINCT` in this example, the fact that I decided to do so anyway gives you a broad hint as to the correct answer to that question. Can you think of an example where `DISTINCT` would be required to avoid duplicates?

5.8 Wrapping and unwrapping in SQL

The theory book describes operators `WRAP` and `UNWRAP` in connection with attributes whose declared types are tuple types. Example 5.14 in that book shows how extension and projection can be used to replace a given set of attribute values in each tuple of a given relation by a single tuple consisting of those values; the next example then illustrates the use of `WRAP` as a convenient shorthand for the same purpose (on the admittedly rare occasions on which it is likely to arise in practice).

The effect of Example 5.14 can be obtained in SQL but note that one needs to write down not only the names of the columns being wrapped but also the names and declared types of the columns not being wrapped.

Example 5.14: Collecting column values together

```
SELECT Name, Phone, Email,
       CAST ( ROW ( House, Street, City, Zip ) AS
              ROW ( House VARCHAR(100), Street VARCHAR(100),
                    City VARCHAR(100), Zip VARCHAR(10) ) )
       AS Address
FROM   CONTACT_INFO
```

As before, we need to use `CAST` because the result of an invocation of `ROW` has unnamed fields. The example assumes, therefore, a definition such as the following for the base table `CONTACT_INFO`:

```
CREATE TABLE CONTACT_INFO ( Name VARCHAR(100) PRIMARY KEY,
                            Phone VARCHAR(15) NOT NULL,
                            Email VARCHAR(50) NOT NULL,
                            House VARCHAR(100) NOT NULL,
                            Street VARCHAR(100) NOT NULL,
                            City VARCHAR(100) NOT NULL,
                            Zip VARCHAR(10) NOT NULL,
                          ) ;
```

SQL has no shorthand similar to `WRAP`, nor for `UNWRAP`. Example 5.15 here shows how unwrapping can be done in longhand in SQL.

Example 5.15: Unwrapping in SQL (inverse of Example 5.14)

Letting `CONTACT_INFO_WRAPPED` denote the result of Example 5.14:

```
SELECT Name,
       Address.House as House,
       Address.Street as Street,
       Address.City as City,
       Address.Zip as Zip
FROM   CONTACT_INFO_WRAPPED
```

`Address.House` in this example is equivalent to **Tutorial D**'s `House FROM Address`. The use of a dot here is consistent with its use with range variables—recall that a range variable also denotes a row, "ranging" over the rows of a table.

5.9 Table Comparison

The theory book includes the following definitions for relation comparisons in **Tutorial D**:

Let *r1* and *r2* be relations having the same heading. Then:

r1 ⊆ *r2* is *true* if every tuple of *r1* is also a tuple of *r2*, otherwise *false*.
r1 ⊇ *r2* is equivalent to *r2* ⊆ *r1*
r1 = *r2* is equivalent to *r1* ⊆ *r2* AND *r2* ⊆ *r1*

The question arises as to whether SQL tables can be similarly compared. SQL does not have direct counterparts of ⊆ and ⊇. It does of course have =, but table expressions cannot be used as comparands. However, as we have seen in Examples 5.6 et seq., the operator `TABLE` has been available since SQL:2003 to derive from a given table expression a value of a multiset type whose element type is a row type. In other words, `( SELECT * FROM` *`t1`* `) = ( SELECT * FROM` *`t2`* `)` is illegal but we can obtain the required effect by writing `TABLE ( SELECT * FROM` *`t1`* `) = TABLE ( SELECT * FROM` *`t2`* `)`. So, to compare two tables, we have to use an operator named `TABLE` to "convert" them from tables into multisets of rows!

To test for every row of *`t1`* being also a row of *`t2`* we could write, for example, `NOT EXISTS ( SELECT * FROM` *`t1`* `EXCEPT SELECT * FROM` *`t2`* `)`. In fact, SQL's `NOT EXISTS` is an exact counterpart of **Tutorial D**'s `IS_EMPTY` operator. However, note carefully that the case where every row in *`t1`* appears in *`t2`* and every row of *`t2`* appears in *`t1`* does not guarantee that *`t1`* and *`t2`* are the same table. Row *r* might appear twice in *`t1`* but only once in *`t2`*, for example.

You should now be able to write SQL counterparts for the theory book's Examples 5.16 and 5.17, so I leave those as exercises for the reader.

5.10 Other Operators on Tables and Rows

Section 5.10 in the theory book covers some of **Tutorial D**'s additional operators involving relations and tuples. If only for the sake of completeness, we need to look for SQL counterparts of these.

Row Membership Test: We have already seen, in Section 5.2, SQL's IN operator, for **Tutorial D**'s ∈ (spelled as `IN` in *Rel*).

Row Extraction

For **Tutorial D**'s `TUPLE FROM` *r*, SQL has *row subqueries*. These are just like scalar subqueries (see Section 5.3) except that they may specify more than one column. For example, when appearing in a suitable context, the expression

```
( SELECT * FROM COURSE WHERE CourseId = CID('C1') )
```

yields the row denoted by

```
CAST ( ROW ( 'C1', 'Database' ) AS
       ROW ( CourseId CID, Title VARCHAR(100) ) )
```

A row subquery whose table operand is empty yields a row in which every field is `NULL`.

Field Extraction: For extracting a field from a row, using dot qualification, see Example 5.15 in Section 5.8.

Row Counterparts of Table Operators

SQL does not have counterparts of **Tutorial D**'s tuple rename, tuple projection, tuple extension, tuple join and tuple compose. To obtain the same effects as these operators on row *r*, one has first to derive the table *t* consisting of just *r*, then apply the SQL counterpart of the corresponding relational operator on *t*, putting parentheses around the table expression so that, so long as the context is appropriate, it becomes a row subquery.

For example, if *r* has fields named `a`, `b`, and `c`, we can simulate a tuple renaming of a tuple projection to obtain the row consisting of just `a` and `b`, with `b` renamed to `x`, by `( SELECT a, x FROM VALUES (`*r*`) AS t(a, x, c) )`. The snag here is that the columns of a `VALUES` expression have implementation-dependent names, so we cannot rely on the field names of *r* being propagated to the table. We therefore have to specify the names in parentheses after the range variable name, `t`. At least that gives us the slight short cut of renaming `b` as `x` on the fly, so to speak.

EXERCISES

1. Does SQL have a counterpart of *r1* `COMPOSE` *r2* when *r1* and *r2* have identical headings? If so, what is it, in general? If not, why not?

2. Write an SQL expression that is equivalent to Example 5.9, repeated below, but does not use a `SELECT` expression in a `FROM` clause.

```
SELECT  CourseId, COALESCE(n, 0) AS n
FROM    COURSE LEFT NATURAL JOIN
      ( SELECT   CourseId, COUNT(*) AS n
        FROM     EXAM_MARK
        GROUP BY CourseId ) AS T
```

3. Write an SQL expression that is equivalent to the example below but does not use a `SELECT` expression in a `FROM` clause.

```
SELECT  CourseId, Title, AvgMark
FROM    COURSE NATURAL JOIN
      ( SELECT   CourseId, AVG(Mark) AS AvgMark
        FROM     EXAM_MARK
        GROUP BY CourseId ) AS T
```

4. In connection with Example 5.13, can you give an example where the same row might appear more than once in the result if `DISTINCT` is omitted? If so, give it; otherwise explain.

5. Using the suppliers-and-parts database shown in Figure 4.13, write SQL expressions for the following queries:

a) Get the total number of parts supplied by supplier S1.

b) Get supplier numbers for suppliers whose city is first in the alphabetic list of such cities.

c) Get part numbers for parts supplied by all suppliers in London.

d) Get supplier numbers and names for suppliers who supply all the purple parts.

e) Get all pairs of supplier numbers, S*x* and S*y* say, such that S*x* and S*y* supply exactly the same set of parts each.

f) Write a truth-valued expression to determine whether all supplier names are unique in `S`.

g) Write a truth-valued expression to determine whether all part numbers appearing in `SP` also appear in `P`.

6. Give SQL counterparts of the theory book's Examples 5.16 and 5.17.

7. Tutorial D's `TUPLE` operator takes a commalist of expressions, each one paired with an attribute name. By contrast, SQL's `ROW` operator takes a commalist of expressions without accompanying field names. What are the advantages and disadvantages of SQL's approach?

8. Distinguish between SQL's table types and its multiset types whose element types are row types.