FROM: "An Introduction to Database Systems", 7th edition, CJ Date, Addison-Wesley, 2000.

An SQL query is formulated as a **table expression**, of potentially considerable complexity. We do not get into all of that complexity here; rather, we simply present a set of examples, in the hope that those examples will highlight the most important points. The examples are based on the SQL table definitions for suppliers and parts shown in Chapter 4 (Fig. 4.1). Recall in particular that there are no user-defined data types in the SQL version of that database; instead, all columns are defined in terms of one of the SQL builtin types. *Note:* A more complete and more formal treatment of SQL expressions in general, and SQL table expressions in particular, appears in Appendix A.

**7.7.1 Get color and city for "nonParis" parts with weight greater than ten pounds.**

```
SELECT PX.COLOR, PX.CITY
FROM   P AS PX
WHERE  PX.CITY <> 'Paris'
AND    PX.WEIGHT > 10.0 ;
```

Points arising:

1. Note the use of the comparison operator "<>" (not equals) in this example. The usual scalar comparison operators are written as follows in SQL: =, <>, <, >, <=, >=.
2. Note also the specification "P AS PX" in the FROM clause. That specification effectively constitutes the definition of a (tuple-calculus-style) range variable called PX, with range the current value of table P. The scope of that definition is, loosely, the table expression in which it appears. *Note:* SQL calls PX a *correlation name*.
3. SQL also supports the notion of *implicit* range variables, according to which the query at hand might equally well have been expressed as follows:

```
SELECT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > 10.0 ;
```

The basic idea is to allow a table name to be used to denote an implicit range variable that ranges over the table in question (provided of course that no ambiguity results). In

---

*One consequence of that growth is that—as noted in the annotation to reference [4.18], *q.v.*—the entire "IN *<subquery>*" construct could now be removed from the language with no loss of functionality! This fact is ironic, since it was that construct that the "Structured" in the original name "Structured Query Language" referred to; indeed, it was that construct that was the original justification for adopting SQL rather than the algebra or the calculus in the first place.

the example, the FROM clause FROM P can be regarded as shorthand for a FROM clause that reads FROM P AS P. In other words, it has to be clearly understood that the "P" in (e.g.) "P.COLOR" in the SELECT and WHERE clauses here does *not* stand for table P—it stands for a *range variable* called P that ranges over the table with the same name.

4. As noted in Chapter 4, we could have used unqualified column names throughout this example, thereby writing:

```
SELECT COLOR, CITY
FROM   P
WHERE  CITY <> 'Paris'
AND    WEIGHT > 10.0 ;
```

   The general rule is that unqualified names are acceptable if they cause no ambiguity. In our examples, however, we will generally include all qualifiers, even when they are technically redundant. Unfortunately, however, there are certain contexts in which column names are explicitly required *not* to be qualified! The ORDER BY clause is a case in point—see the example immediately following.

5. The **ORDER BY** clause, mentioned in connection with DECLARE CURSOR in Chapter 4, can also be used in interactive SQL queries. For example:

```
SELECT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > 10.0
ORDER  BY CITY DESC ;
```

6. We remind you of the "SELECT *" shorthand, also mentioned in Chapter 4. For example:

```
SELECT *
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > 10.0 ;
```

   The star in "SELECT *" is shorthand for a commalist of all column names in the table(s) referenced in the FROM clause, in the left-to-right order in which those column(s) are defined within those table(s). We remark that the star notation is convenient for interactive queries, since it saves keystrokes. However, it is potentially dangerous in embedded SQL—i.e., SQL embedded in an application program—because the meaning of the "*" might change (e.g., if a column is added to or dropped from some table, via ALTER TABLE).

7. *(Much more important than the previous points!)* Note that, given our usual sample data, the query under discussion will return *four* rows, not two, even though three of those four rows are identical. SQL does not eliminate redundant duplicate rows from a query result unless the user explicitly requests it to do so via the keyword **DISTINCT**, as here:

```
SELECT DISTINCT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > 10.0 ;
```

This query will return two rows only, not four.

It follows that *the fundamental data object in SQL is not a relation*—it is, rather, a table, and SQL-style tables contain (in general) not sets but *bags* of rows. (A "bag"—also called a *multi-set*—is like a set but permits duplicates.) SQL thus violates *The Information Principle* (see Chapter 3, Section 3.2). One consequence is that the fundamental operators in SQL are not true relational operators but bag analogs of those operators; another is that results and theorems that hold true in the relational model—regarding the transformation of expressions, for example [5.6]—do not necessarily hold true in SQL.

**7.7.2 For all parts, get the part number and the weight of that part in grams.**

The specification AS GMWT introduces an appropriate result column name for the "computed column." The two columns of the result table are thus called P# and GMWT, respectively. If the AS clause had been omitted, the corresponding result column would effectively have been unnamed. Observe, therefore, that SQL does not actually require the user to provide a result column name in such circumstances, but we will always do so in our examples.

**7.7.3 Get all combinations of supplier and part information such that the supplier and part in question are colocated.** SQL provides many different ways of formulating this query. We give three of the simplest here.

1.

2.

3. S NATURAL JOIN P ;

The result in each case is the **natural join** of tables S and P (on cities).

The first of the foregoing formulations—which is the only one that would have been valid in SQL as originally defined (explicit JOIN support was added in SQL/92)—merits further discussion. Conceptually, we can think of that version of the query as being implemented as follows:

- First, the FROM clause is executed, to yield the Cartesian product S TIMES SP. (Strictly, we should worry here about renaming columns before we compute the product. We ignore this issue for simplicity. Also, recall that—as we saw in Exercise 6.12 in Chapter 6—the "Cartesian product" of a single table $T$ can be regarded as just $T$ itself.)

- Next, the WHERE clause is executed, to yield a **restriction** of that product in which the two CITY values in each row are equal (in other words, we have now computed the *equijoin* of suppliers and parts over cities).
- Finally, the SELECT clause is executed, to yield a **projection** of that restriction over the columns specified in the SELECT clause. The final result is the natural join.

Loosely speaking, therefore, FROM in SQL corresponds to Cartesian product, WHERE to restrict, and SELECT to project, and the SQL SELECT–FROM–WHERE represents a projection of a restriction of a product. See Appendix A for further discussion.

**7.7.4 Get all pairs of city names such that a supplier located in the first city ships a part stored in the second city.**

**7.7.5 Get all pairs of supplier numbers such that the two suppliers concerned are colocated.**

Explicit range variables are clearly required in this example. Note that the introduced column names SA and SB refer to columns of the *result table*, and so cannot be used in the WHERE clause.

**7.7.6 Get the total number of suppliers.**

The result here is a table with one column, called N, and one row, containing the value 5. SQL supports the usual aggregate operators **COUNT, SUM, AVG, MAX,** and **MIN,** but there are a few SQL-specific points the user needs to be aware of:

- In general, the argument can optionally be preceded by the keyword DISTINCT—as in, e.g., SUM (DISTINCT QTY)—to indicate that duplicates are to be eliminated before the aggregation is applied. For MAX and MIN, however, DISTINCT is irrelevant and has no effect.
- The special operator COUNT(*)—DISTINCT not allowed—is provided to count all rows in a table without any duplicate elimination.

- Any nulls in the argument column (see Chapter 18) are eliminated before the aggregation is done, regardless of whether DISTINCT is specified, except for the case of COUNT(*), where nulls behave as if they were values.
- If the argument happens to be an empty set, COUNT returns zero; the other operators all return null. (This latter behavior is logically incorrect—see reference [3.3]—but it is the way SQL is defined.)

**7.7.7 Get the maximum and minimum quantity for part P2.**

```
                                                    MINQ
```

Observe that the FROM and WHERE clauses here both effectively provide part of the argument to the two aggregate operators. They should therefore logically appear within the argument-enclosing parentheses. Nevertheless, the query is indeed written as shown. This unorthodox approach to syntax has significant negative repercussions on the structure, usability, and orthogonality* of the SQL language. For instance, one immediate consequence is that aggregate operators cannot be nested, with the result that a query such as "Get the average total-part-quantity" cannot be formulated without cumbersome circumlocutions. To be specific, the following query is *** *ILLEGAL* ***:

```
SELECT AVG ( SUM ( SP.QTY ) )              -- Warning! Illegal!
FROM   SP ;
```

Instead, it has to be formulated something like this:

```
SELECT AVG ( X )
FROM ( SELECT SUM ( SP.QTY ) AS X
       FROM   SP
       GROUP  BY SP.S# ) AS POINTLESS ;
```

See the example immediately following for an explanation of GROUP BY, and several subsequent examples for an explanation of "nested subqueries." It is worth noting that the ability to nest a subquery inside the FROM clause, as here, was new with SQL/92 and is still not widely implemented. *Note:* The specification AS *POINTLESS* is pointless but is required by SQL's syntax rules (see Appendix A).

**7.7.8 For each part supplied, get the part number and the total shipment quantity.**

*Orthogonality means *independence*. A language is orthogonal if independent concepts are kept independent, not mixed together in confusing ways. Orthogonality is desirable because the less orthogonal a language is, the more complicated it is and—paradoxically but simultaneously—the less powerful it is.

The foregoing is the SQL analog of the relational algebra expression

```
SUMMARIZE SP PER SP { P# } ADD SUM ( QTY ) AS TOTQTY
```

or the tuple calculus expression

```
( SPX.P#, SUM ( SPY WHERE SPY.P# = SPX.P#, QTY ) AS TOTQTY )
```

Observe in particular that if the GROUP BY clause is specified, expressions in the SELECT clause must be **single-valued per group.**

Here is an alternative (actually preferable) formulation of the same query:

```
SELECT P.P#, ( SELECT SUM ( SP.QTY )
               FROM   SP
               WHERE  SP.P# = P.P# ) AS TOTQTY
FROM   P ;
```

The ability to use nested subqueries to represent scalar values (e.g., within the SELECT clause, as here) was added in SQL/92 and represents a major improvement over SQL as originally defined. In the example, it allows us to generate a result that includes rows for parts that are not supplied at all, which the previous formulation (using GROUP BY) does not. (The TOTQTY value for such parts will unfortunately be given as null, however, not zero.)

**7.7.9 Get part numbers for parts shipped by more than one supplier.**

The HAVING clause is to groups what the WHERE clause is to rows; in other words, HAVING is used to eliminate groups, just as WHERE is used to eliminate rows. Expressions in a HAVING clause must be single-valued per group.

**7.7.10 Get supplier names for suppliers who supply part P2.**

*Explanation:* This example makes use of a subquery in the WHERE clause. Loosely speaking, a **subquery** is a SELECT–FROM–WHERE–GROUP BY–HAVING expression that is nested somewhere inside another such expression. Subqueries are used among other things to represent the set of values to be searched via an **IN condition,** as the example illustrates. The system evaluates the overall query by evaluating the subquery first (at least conceptually). That subquery returns the set of supplier *numbers* for suppliers who supply part P2, namely the set {S1,S2,S3,S4}. The original expression is thus equivalent to the following simpler one:

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# IN ( 'S1', 'S2', 'S3', 'S4' ) ;
```

It is worth pointing out that the original problem—"Get supplier names for suppliers who supply part P2"—can equally well be formulated by means of a *join*, e.g., as follows:

**7.7.11 Get supplier names for suppliers who ship at least one red part.**

Subqueries can be nested to any depth. *Exercise:* Give some equivalent join formulations of this query.

**7.7.12 Get supplier numbers for suppliers with status less than the current maximum status in the S table.**

This example involves *two distinct implicit range variables*, both denoted by the same symbol "S" and both ranging over the S table.

**7.7.13 Get supplier names for suppliers who ship part P2.** *Note:* This example is the same as Example 7.7.10; we show a different solution, in order to introduce another SQL feature.

*Explanation:* The SQL expression "EXISTS (SELECT ... FROM ...)" evaluates to *true* if and only if the result of evaluating the "SELECT ... FROM ..." is not empty. In other words, the SQL **EXISTS** *operator* corresponds to the *existential quantifier* of the tuple calculus (but see reference [18.6]). *Note:* SQL refers to the subquery in this particular example as a *correlated* subquery, since it includes references to a range variable—namely, the

implicit range variable S—that is defined in the outer query. Refer back to the "preferable formulation" of Example 7.7.8 for another example of a correlated subquery.

**7.7.14 Get supplier names for suppliers who do not ship part P2.**



**7.7.15 Get supplier names for suppliers who ship all parts.**



SQL does not include any direct support for the universal quantifier FORALL; hence, "FORALL" queries typically have to be expressed in terms of existential quantifiers and double negation, as in this example.

By the way, it is worth pointing out that expressions such as the one just shown, daunting though they might appear at first glance, are easily constructed by a user who is familiar with relational calculus, as explained in reference [7.4]. Alternatively—if they are still thought too daunting—then there are several "workaround" approaches that can be used that avoid the need for negated quantifiers. In the example, for instance, we might write:

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  ( SELECT COUNT ( SP.P# )
         FROM   SP
         WHERE  SP.S# = S.S# ) =
       ( SELECT COUNT ( P.P# )
         FROM   P ) ;
```

("Get names of suppliers where the count of the parts they supply is equal to the count of all parts"). Note, however, that:

- First, this latter formulation relies—as the NOT EXISTS formulation did not—on the fact that every shipment part number is the number of some existing part. In other words, the two formulations are equivalent (and the second is correct) only because a certain *integrity constraint* is in effect (see the next chapter).
- Second, the technique used in the second formulation to compare the two counts (subqueries on both sides of the equals sign) was not supported in SQL as originally defined but was added in SQL/92. It is still not supported in many products.
- We remark too that what we would really like to do is to compare two *tables* (see the discussion of relational comparisons in Chapter 6), thereby expressing the query as follows:

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  ( SELECT SP.P#
         FROM   SP
         WHERE  SP.S# = S.S# ) =
       ( SELECT P.P#
         FROM   P ) ;
```

SQL does not directly support comparisons between tables, however, and so we have to resort to the trick of comparing table *cardinalities* instead (relying on our own external knowledge to ensure that if the cardinalities are the same then the tables are the same too, at least in the situation at hand). See Exercise 7.11 at the end of the chapter.

**7.7.16 Get part numbers for parts that either weigh more than 16 pounds or are supplied by supplier S2, or both.**

Redundant duplicate rows are always eliminated from the result of an unqualified **UNION**, **INTERSECT**, or **EXCEPT** (EXCEPT is the SQL analog of our MINUS). However, SQL also provides the qualified variants **UNION ALL**, **INTERSECT ALL**, and **EXCEPT ALL**, where duplicates (if any) are retained. We deliberately omit examples of these variants.

This brings us to the end of our list of retrieval examples. The list is rather long; nevertheless, there are numerous SQL features that we have not even mentioned. The fact is, SQL is an extremely *redundant* language [4.18], in the sense that it almost always provides numerous different ways of formulating the same query, and space simply does not permit us to describe all possible formulations and all possible options, even for the comparatively small number of examples we have discussed in this section.