The Essence of SQL:

A Guide to Learning Most of SQL in the Least Amount of Time

by David Rozenshtein

THE ESSENCE OF SQL:

A Guide to Learning Most of SQL in the Least Amount of Time

David Rozenshtein

Department of Computer Science
Long Island University

40087 Mission Boulevard, Suite 167, Fremont, CA 94539
(800) 943-9300 Voice
(510) 656-5116 Fax
http://www.sqlforum.com

Library of Congress Catalog Card Number 96-092799
ISBN 0-9649812-1-1

Editor: Tom Bondur
Cover concept: Anna Bondur
Cover design: Tom McKeith
Inside layout: Mimi Fujii

40087 Mission Boulevard, Suite 167, Fremont, CA 94539
(800) 943-9300 Voice
(510) 656-5116 Fax
http://www.sqlforum.com

Contents

List of Figures

1 Introduction

This essay is dedicated to the proposition that one can often accomplish 80% of the task in 20% of the time, or, in this case, that one can learn most of the important features and capabilities of SQL quickly. To achieve this, we have developed a new approach to presenting SQL.

Traditionally, SQL is presented in a "bottom-up" fashion, by enumerating all of its features in some order and illustrating each feature with an example query. While complete in its treatment of the language, this approach often fails to clearly distinguish between what is essential and what is secondary in the language, and frequently leaves an impression of too long a "laundry list" of language features. Even more importantly, by concentrating on the language first and motivation second, this approach does not clearly explain *why* certain features are present in the language, *when* they should be used, nor *how* to actually pose queries using SQL.

Our approach, on the other hand, is "top-down." We begin by identifying a rather short list of standard questions, or more precisely types of questions, that are often asked of relational databases. We then show how these standard questions are posed in SQL, introducing and motivating the use of its capabilities and features as they become relevant.

As a result, we develop concrete solutions for a set of well specified question types. Solutions for particular business problems can then be developed by selecting the appropriate question type or combination of types and rephrasing the corresponding standard solutions to fit the particular problem domain.

It is important to note that we do not claim completeness in our approach— some features of the language will not be covered here. We also do not claim objectivity in the selection of features. This essay is based on the SQL working and teaching experience of the author, and reflects his views on the language.

While we assume a familiarity with the basic relational concepts of tables, attributes, keys, etc., no prior knowledge of SQL itself is assumed. In fact, those who already know some SQL should set their knowledge aside and start anew with this essay.

Finally, whatever liberties we take in this essay are all motivated by its primary purpose — to leave the reader with a deep understanding of the essence of SQL and a set of cookbook recipes for its immediate use.

2 Example Database

All of the SQL queries in this essay will be posed with respect to the example database shown in Figure 2.1. This database consists of five tables, three of which correspond to entity classes: Student, Course and Professor, and two to relationships: Take— between students and courses, and Teach— between professors and courses.

Student(Sno, Sname, Age)
Course(Cno, Title, Credits)
Professor(Fname, Lname, Dept, Rank, Salary, Age)
Take(Sno, Cno, Grade)
Teach(Fname, Lname, Cno)

Figure 2.1: Example database.

In the tables, the column name abbreviations have the following meaning:

Sno:	student number;
Sname:	student name;
Cno:	course number;
Fname:	professor's first name;
Lname:	professor's last name;
Dept:	professor's department.

Relational keys are identified by underlining. Note that table Professor has a composite— two column—key. We have split professors' names into separate first and last names to motivate certain SQL features introduced later.

3 A Bird's Eye View of SQL

SQL can be divided into four parts, or *sublanguages*, as they are traditionally called: *data definition language* (DDL), *data manipulation language* (DML), *system administration language* (SAL) and *query language*.

DDL provides facilities for creating and destroying tables and indices, as well as for affecting their physical layout. A simple example of DDL, creating our table Student and its related index is shown in Figure 3.1. (In this essay, we use generic SQL syntax which, except for the data type names and minor syntactic differences, is applicable to all SQL dialects.)

```
CREATE TABLE Student(
    Sno         char(9)          NOT NULL,
    Sname       varchar(20)      NULL,
    Age         int              NULL)

CREATE UNIQUE INDEX Student_ndx ON
    Student(Sno)
```

Figure 3.1: DDL for table Student.

The table creation syntax closely mimics record layout definitions in most modern 3GLs. Concept of NULL and the meaning of the NULL and NOT NULL qualifiers will be explained later.

The index creation syntax is also straightforward— the name of the index is Student_ndx, and it is defined on table Student with respect to column Sno. Furthermore, because

of the UNIQUE qualifier, this index will enforce the uniqueness of Sno values in the Student table. While the stated purpose of indices is to speed up query execution, in many systems UNIQUE indices also provide the only way to enforce uniqueness of key values.

We note that details of physical layout do not shed any light on the "meaning" of SQL and are thus omitted from this example and this essay.

Destruction of tables and indices is done using keyword DROP, as in

```
DROP TABLE Student
```

DML provides facilities to insert, delete and modify rows in tables. These facilities are best presented after the query facilities of SQL, and are left for the end of this essay.

SAL provides facilities for managing the system— e.g., for setting up security and authorization schemes for the database. While essential for system administration tasks, these features again do not shed any light on SQL meaning, and are omitted from this essay.

The query language component of SQL provides facilities for asking questions about the data. These query facilities make SQL what it is, and this essay is devoted specifically to them.

Roughly speaking, all SQL query facilities can be divided into six categories, as shown in Figure 3.2.

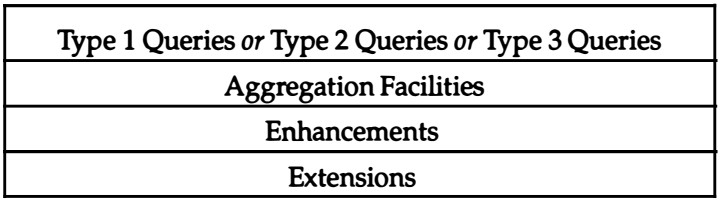

Figure 3.2: SQL query types.

Types 1, 2 and 3 are the three distinct query types, each motivated by a different category of questions and database designs. For those already familiar with SQL, Type 1 queries correspond to "flat" (single level) queries, and Type 2 and Type 3 queries correspond to "nested" (multi-level) queries: Type 2— without correlations, and Type 3— with correlations.

Aggregation facilities, enhancements and extensions are orthogonal to these three query types, and can be used in conjunction with all of them. They can also be used in combinations with each other.

Aggregation facilities provide means for computing various aggregates, such as sums and averages, and for grouping the data.

Enhancements are those features that make SQL more practical, but do not fundamentally increase its expressive power. For example, ability to sort the answer is one such enhancement.

Extensions encompass various advanced features— most important of which is the explicit WHILE loop— that have been added to SQL recently by various vendors. These

extensions make the language fundamentally more powerful; however, they are still quite non-standard, and are considered only briefly at the end of this essay.

4 The Standard Questions

We begin our presentation of SQL with a list of standard questions, as shown in Figure 4.1. (Designation E1 stands for "English question 1," etc.) This list will be extended with additional standard questions as we proceed. The key subphrases that make these questions standard are italicized for emphasis.

E1. Who takes (the course with the course number) CS112?
(By "Who" we mean that we want the student numbers retrieved. If we want the names retrieved, we will explicitly say so.)
E2. What are student numbers and names of students who take CS112?
E3. Who takes CS112 *or* CS114?
E4. Who takes *both* CS112 *and* CS114?
E5. Who *does not* take CS112?
E6. Who takes a course which *is not* CS112?
E7. Who takes *at least* 2 courses (i.e., at least 2 courses with different course numbers)?
(A more general question: Who takes at least 3, 4, 5, etc., courses?)
E8. Who takes *at most* 2 courses?
(More generally: 3, 4, 5, etc., courses?)
E9. Who takes *exactly* 2 courses?
(More generally: 3, 4, 5, etc., courses?)
E10. Who takes *only* CS112?
E11. Who takes *either* CS112 *or* CS114?
E12. Who are the *youngest* students?
(Similarly, Who are the *oldest* students?)
E13. Who takes *every* course?

Figure 4.1: A list of standard questions.

---

5 The First Two Questions

The first two questions fall into the category of Type 1 queries. All Type 1 SQL queries have the basic form

SELECT <list of desired columns>
FROM <list of tables>
WHERE <Boolean condition>

where SELECT, FROM and WHERE are keywords designating the three basic components, or *clauses*, of the query. (While SQL is generally case insensitive, for clarity we will show all keywords in upper case.)

In thinking about, understanding and composing SQL queries, we look at the FROM clause first. This clause lists the table(s) that need to be considered to answer the question. (To determine this list, pretend that you have to answer this question without a computer system, using just paper copies of the tables, and think of which of them you would actually need.)

In question E1— *Who takes CS112?*— "Who" stands for student numbers (Sno), and CS112 stands for a course number (Cno). Thus, since all relevant information is contained in table Take, the FROM clause becomes

FROM Take

Next, we consider the WHERE clause. It contains a Boolean condition which defines which rows from table(s) in the FROM clause should be retrieved by the query. For question E1 this condition is for the course number to be CS112, thus making the WHERE clause

WHERE (Cno = "CS112")

(Whether one uses single or double quotes around string literals is usually system dependent. Also, while parenthesis around conditions are often not required, we use them to improve readability.)

Finally, the SELECT clause lists the column(s) that define the structure of the answer. Since in this case we just want the student numbers retrieved, the SELECT clause becomes

```
SELECT Sno
```

The final query then takes the form shown in Figure 5.1.

```
SELECT Sno
FROM Take
WHERE (Cno = "CS112")
```

Figure 5.1: Query Q1 for question E1 – Who takes CS112?

While it is true that this query does intuitively correspond to question E1, intuition is not a reliable means for understanding SQL. So, to be safe in interpreting SQL queries, we introduce a conceptual device known as the *query evaluation mechanism.*

A query evaluation mechanism gives us a precise and formal algorithm to trace queries.

> *This ability to trace is fundamental to all SQL programming, since it is only by following the trace that we can see how the answer is actually computed, and thus understand its true meaning.*

Different types of SQL queries have different evaluation mechanisms. The Type 1 evaluation mechanism is shown in Figure 5.2

1. Take a *cross-product* of all tables in the FROM clause — i.e., create a temporary table consisting of all possible combinations of rows from all of the tables in the FROM clause. (If the FROM clause contains a single table, skip the cross-product and just use the table itself.)
2. Consider every row from the result of Step 1 *exactly once*, and evaluate the WHERE clause condition for it.
3. If the condition returns True in Step 2, formulate a resulting row according to the SELECT clause, and retrieve it.

Figure 5.2: The Type 1 SQL evaluation mechanism.

Given this evaluation mechanism, we can now formally trace query Q1, as follows.

1. Since query Q1 involves only one table, we effectively skip Step 1.
2. We look at every row from table Take, and evaluate condition (Cno = "CS112") for it.
3. We take the Sno values from those rows where this condition returns True, and place them into the result.

By following this trace, we can now confidently assert that query Q1 indeed corresponds to our question E1. (Do not be deceived into thinking that just because this evaluation mechanism and the consequent trace are so simple, using the mechanism is the same as just using the basic intuition. As we will demonstrate very shortly, this is not at all so.)

We note that because evaluation mechanisms are formal devices, they do not represent actual evaluation strategies taken by real systems. (No real system would be caught dead actually taking cross-products all the time.) All that is required of them is that they always give *the same result as* a real system. Thus, in developing this particular form of the evaluation mechanism, we were not concerned with its apparent inefficiency, and concentrated instead on its clarity and ease of use in tracing.

The SQL query corresponding to question E2— *What are student numbers and names of students who take CS112?*— is shown in Figure 5.3.

```
SELECT Student.Sno, Sname
FROM Student, Take
WHERE (Student.Sno = Take.Sno)
   AND (Cno = "CS112")
```

Figure 5.3: Query Q2 for question E2 – What are student numbers and names of students who take CS112?

This query shows an example of using several tables in the FROM clause and several columns in the SELECT clause. It also introduces a new syntactic feature.

Because cross-products retain all columns from all of the tables involved, in this case the result of the cross-product will have two columns labeled Sno— one from the Student table and the other from the Take table. Thus, every time we refer to Sno, we need to explicitly specify, or *disambiguate*, which of the two Sno columns we mean. The use of the "dotted" notation Student.Sno and Take.Sno achieves this. This notation (which is quite standard in most programming languages for use with record variables and their

fields) is called a *prefix* notation in SQL, with "Student." and "Take." called prefixes.

Also, while it does not matter which of the two Sno columns— Student.Sno or Take.Sno— we choose for the answer (after all, they are equal to each other), we must explicitly specify one of them in the SELECT clause. Not doing so would cause a syntax error. (An informal explanation for this is as follows: Column name disambiguation is a syntactic issue and must be resolved at compile-time, while equality is not known until run-time.)

We now trace this query, using our evaluation mechanism, as follows. (To better follow this trace, make some example data for the tables, and execute its steps on paper.)

1. We take the cross-product of tables Student and Take. This cross product would contain every combination of rows from these tables. (A good way to visualize the result of this cross-product is to think of it as comprised of "wide" rows formed by "concatenating" the Student and Take rows from each combination.)
2. We evaluate the WHERE clause condition for every such combination. This condition has two parts, with the conjunct (Student.Sno = Take.Sno) assuring that the Student row and the Take row in the combination deal with the same student, and the conjunct (Cno = "CS112") assuring that we are dealing with course CS112.
3. For those combinations where the condition returns True, we choose the Sno and

Sname values from the Student row and put them into the answer.

Again, it is by following this trace, that we can confidently conclude that query Q2 indeed corresponds to question E2.

While required to disambiguate which columns come from which tables when there are several same-named columns in the result of the cross-product, the prefix notation is always permitted for all column references. Thus, even though it is not necessary to use a prefix with Sname or Cno— there is only one column called Sname and only one column called Cno in the cross-product, it is perfectly legal to write our query Q2 as shown in Figure 5.4.

```
SELECT Student.Sno, Student.Sname
FROM Student, Take
WHERE  (Student.Sno = Take.Sno)
    AND (Take.Cno = "CS112")
```

Figure 5.4: Query Q2 rewritten with all explicit prefixes.

The essential feature of the Type 1 evaluation mechanism is that every wide row from the result of the cross-product is considered *exactly once*. Thus, even though the order in which these rows are considered is not specified, fundamentally, Type 1 queries involve only *one pass* through the data.

This means that the decision of whether or not some row combination from the cross-product will contribute to the answer has to be made exactly when this combination is considered, and not at any other time during the evaluation. In particular, it cannot be delayed until after some other row combinations have been looked at. As we will see shortly, this one-pass-only property of Type 1 queries will be of great importance.

6 Standard Questions Involving "or" and "both-and"

Our standard question E3— *Who takes CS112 or CS114?*— is posed by query Q3 shown in Figure 6.1.

```
SELECT Sno
FROM Take
WHERE (Cno = "CS112")
      OR (Cno = "CS114")
```

Figure 6.1: Query Q3 for question E3 – Who takes CS112 or CS114?

A trace of this query shows its correctness. The only new issue here is this: If some student actually does take both CS112 and CS114, then his Sno will be retrieved twice by the query— the first time when the query processes a Take row with his Sno and CS112, and the second time when it processes a Take row with his Sno and CS114. Such duplicates can be eliminated by rewriting the SELECT clause as follows:

```
SELECT DISTINCT Sno
```

The keyword DISTINCT, which eliminates duplicate rows from the answer, is one of the enhancements available in SQL. (It is considered an enhancement because it does not change the meaning of the answer— it simply makes it more concise.) However, its use is expensive, since duplicate elimination requires the answer to be sorted. Thus, it should be used only when concerns for the clarity of the answer outweigh those for its efficiency.

An interesting variation of this query is to rewrite the WHERE clause as

```
WHERE (Cno IN ("CS112", "CS114"))
```

which uses the *list membership* operator IN. (Operator IN returns True if the value on its left is equal to one of the values in the list on its right.) While just providing an alternative formulation in this case, this operator is actually a significant feature of SQL and will become necessary when we introduce Type 2 queries.

Our standard question E4— *Who take both CS112 and CS114?*— presents a much more interesting case. First, consider the query of Figure 6.2.

```
SELECT Sno
FROM Take
WHERE (Cno = "CS112")
   AND (Cno = "CS114")
```

Figure 6.2: First attempt at question E4 – Who takes both CS112 and CS114?

This query, which was generated from query Q3 by replacing OR with AND, reflects a misplaced intuition that logical operators directly model, and thus can be automatically substituted for, their English counterparts. However, intuition is not a reliable tool when programming in SQL, and the query of Figure 6.2 is not a correct implementation of question E4. It will compile, however, and therefore will generate some answer. (Before proceeding further, try to figure out exactly what answer this query will generate, and why it is incorrect.)

Recall that the conditions of Type 1 queries are evaluated on a row-by-row basis, once per row. Since no row can have a Cno value which is *simultaneously* equal to "CS112" and to "CS114", the condition of this query will always return False. Thus, the answer to this query will always be empty.

While question E4 can be posed as a Type 1 query, to discover this solution, we need to employ a bit of "reverse engineering." In other words, we first need to figure out how we can actually compute the answer to this question in a manner consistent with the Type 1 evaluation mechanism— in effect, visualizing the corresponding trace, and then to "back into" the SQL query itself. We note that:

> *This ability to visualize traces and then reverse engineer the appropriate SQL code is the key to one's mastery of SQL.*

To visualize the trace, we first need to determine which table(s) need to be considered. Since the question *Who takes both CS112 and CS114?* involves only student numbers (Sno) and course numbers (Cno), the only table that is needed is table Take.

However, as we have just argued, since every Take row contains only one Cno value, and since it is considered only once by the evaluation mechanism, we cannot evaluate our condition directly on the rows of Take. What we need instead is a table with rows that for each student would list not one, *but two*, course numbers for the courses he takes.

But how can we generate such a table from our basic table Take? The answer is: In two steps. First, we can compute a cross-product of table Take with itself. (Think of making a copy of Take and using it in the cross-product with the original.)

Second, we can impose equality on the two Sno values in each wide row in the result of this cross-product. This would remove those wide rows where the Sno value from the first copy of Take is different from the Sno value from the second copy.

Each remaining wide row would then refer to just a single student, and the two— one from each copy of Take— Cno values for the courses he takes. We can then test *one of them* to be "CS112" and *the other* to be "CS114," and for those rows where these tests would succeed, retrieve the Sno value into the answer.

Since in Type 1 SQL queries cross-products are caused by listing tables in the FROM clause, we can informally sketch a query that would have such a behavior as shown in Figure 6.3.

```
SELECT Take.Sno
FROM Take, Take
WHERE  (Take.Sno = Take.Sno)
    AND (Take.Cno = "CS112")
    AND (Take.Cno = "CS114")
```

Figure 6.3: A sketch of the query for E4 – Who takes both CS112 and CS114?

There is an obvious problem with this sketch, however. Since both tables in the FROM clause have the same name (one of them, after all, is a copy of the other), the use of table names for prefixes is not sufficient to disambiguate column references.

What we really want to say here is that in the condition (Take.Sno = Take.Sno) the left Take.Sno is taken from the

first copy of Take, and the right Take.Sno is taken from the second copy of Take. Similarly, in the condition (Take.Cno = "CS112") we mean the first copy of Take and in the condition (Take.Cno = "CS114") we mean the second one. Finally, since both Sno values are the same in the wide row, it does not matter to us which one is retrieved in the SELECT clause, but we must explicitly choose one or the other. None of this, however, is reflected in our query sketch.

To take care of such cases, SQL allows any table in the FROM clause to be optionally followed by its temporary *alias*— a new table name, which is valid only for the duration of the particular query and which is used for prefixes. The aliasing feature allows us to phrase the correct query for question E4 as shown in Figure 6.4.

```
SELECT X.Sno
FROM Take X, Take
WHERE  (X.Sno = Take.Sno)
    AND (X.Cno = "CS112")
    AND (Take.Cno = "CS114")
```

Figure 6.4: Query Q4 for question E4 – Who takes both CS112 and CS114?

Before proceeding further, we want to make several comments regarding the use of aliases. First, the choice of alias names is completely arbitrary as long as they do not conflict with each other or with any real table name used in a query.

Second, aliases are *opaque*— a technical term meaning that an alias completely covers and hides the name of the underlying table. Thus, from the point of view of the SELECT and WHERE clauses, the query of Figure 6.4 involves two tables: one named X and the other named Take.

Third, it is always permitted to alias tables— even if not really necessary. Indeed, to save on typing, SQL programmers often use short aliases for long, meaningful table names given to them by well-meaning database designers. Consequently, our query Q4 could also have been written as shown in Figure 6.5. We stress, however, that using prefix "Take." anywhere in this query would now be syntactically incorrect.

```
SELECT X.Sno
FROM Take X, Take Y
WHERE  (X.Sno = Y.Sno)
    AND (X.Cno = "CS112")
    AND (Y.Cno = "CS114")
```

Figure 6.5: Q4 rewritten with both copies of Take aliased.

Finally, we note that aliases are an essential feature of SQL. Without them, standard questions involving the *both-and* construct could not have been asked as Type 1 queries.

7 Negation in SQL

Negation is one of the most interesting and complex issues in SQL. Both question E5— *Who does not take CS112?*, and question E6— *Who takes a course which is not CS112?* involve negation.

It is important to understand that these are not restatements of each other, but are indeed different questions. The following two example situations, for which these two questions would generate different answers, illustrate this point. (Try to find them before proceeding.)

The first example involves a student who takes both CS112 and some other course, say CS113. Because this student takes CS112, he will not be in the answer to question E5. However, because he takes a course— CS113, in this case— which is not CS112, he will be in the answer to question E6.

The second example involves a student who does not take any courses. (Or, to put it another way, who takes nothing.) Since he does not take any courses, it follows that he does not take CS112; therefore, he will be in the answer to question E5. However, he will not be in the answer to question E6, which requires him to take some course. (Note that the student number of this student would be included in table Student, but would be absent from table Take. For those familiar with the concepts of *referential integrity* and *foreign key constraints*, this situation is perfectly legal and would be allowed in this database.)

Not only are questions E5 and E6 different, they are *orthogonal* to each other— i.e., all four combinations of Yes/No answers for these two questions are possible, as shown in Figure 7.1.

Situation	Included in answer to E5	Included in answer to E6
Student takes both CS112 and CS113	No	Yes
Student takes nothing	Yes	No
Student takes only CS112	No	No
Student takes only CS113	Yes	Yes

Figure 7.1: Orthogonality of questions E5 and E6.

The query posing question E6— *Who takes a course which is not CS112?*— is shown in Figure 7.2.

```
SELECT Sno
FROM Take
WHERE (Cno != "CS112")
```

Figure 7.2: Query Q6 for question E6 – Who takes a course which is not CS112?

A trace of this query shows its correctness. We only note that the answer to this query may contain duplicates. To eliminate them we would need DISTINCT in the SELECT clause.

This is a good time to mention that in addition to equality (=) and inequality (which in some SQL dialects is expressed as != and in some as <>), SQL supports all other standard binary comparators: <, <=, >, and >=. All SQL dialects support these latter comparators for numeric data types; most also support them for strings and other data types as well.

Question E5 — *Who does not take CS112?*— is much trickier. First, note that rewriting the WHERE clause as

```
WHERE NOT (Cno = "CS112")
```

will not work, because conditions NOT (Cno = "CS112") and (Cno != "CS112") are equivalent to each other, and thus would both pose question E6. (We note that SQL supports full Boolean logic including DeMorgan's and standard Boolean Distributive and Associative Laws.)

A more fundamental problem with question E5 emerges when one considers the basic one-pass nature of Type 1 queries. As it turns out, the answer to this question cannot be computed in a single pass, even if we do take a cross-product of tables Student and Take. (Try it on some data examples, and observe that one pass is not enough.)

What is required, instead, are *two passes:* the first pass through table Take to select those students who *do take* CS112, and the second pass through table Student to "get rid of" them. Furthermore, the first pass has to be completed— i.e., we have to identify and in a way "collect" all of these "bad students," before starting on the second pass— to eliminate them.

Since the Type 1 SQL evaluation mechanism is fundamentally "one pass," it is *impossible* to pose question E5 as a Type 1 query, given this or any other reasonable database design. Instead, question E5 requires *subqueries*, which are provided by Type 2 and 3 queries, and which are covered later.

An important general difference between questions E5 and E6 is that E5 involves "real negation" and E6 involves the so called "pseudo-negation." Questions involving pseudo-negation can be posed as Type 1 queries; questions involving real negation cannot.

One way to distinguish between the two is to see *what* is being negated— some *noun constant* (e.g., "is not course CS112" in E6) or a *verb* (e.g., "does not take" in E5). Also, question Q6 can be rephrased by replacing the English word "not" with the phrase "different from" (or "other than"), as in *Who takes some course different from CS112?* This technique is a rule of thumb that works most of the time.

8 Standard Questions Involving "at least"

Question E7— *Who takes at least 2 courses?*— is the last of our standard questions that can be posed as a Type 1 query. The query corresponding to this question, shown in Figure 8.1, is quite similar to the "both-and" query Q4, and has a very similar trace. (An alternative way to pose this query is to replace the inequality in the WHERE clause by the less-than comparison (X.Cno < Take.Cno), assuming that the particular SQL dialect allows less-than comparisons between strings.) To eliminate duplicates from the answer we would need DISTINCT in the SELECT clause.

```
SELECT X.Sno
FROM Take X, Take
WHERE  (X.Sno = Take.Sno)
    AND (X.Cno != Take.Cno)
```

Figure 8.1: Query Q7 for question E7 – Who takes at least 2 courses?

Of course, as noted in our standard questions list, there are many cases of the use of "at least." So, how would we pose the question *Who takes at least 3 courses?* The answer is: The same way, only using 3 copies of table Take, as shown in Figure 8.2.

```
SELECT X.Sno
FROM Take X, Take Y, Take
WHERE  (X.Sno = Y.Sno)
    AND (Y.Sno = Take.Sno)
    AND (X.Cno != Y.Cno)
    AND (Y.Cno != Take.Cno)
    AND (X.Cno != Take.Cno)
```

Figure 8.2: The query for question: Who takes at least 3 courses?

Again a trace shows the correctness of this query. We note that, because equality is transitive, two equalities on Sno are sufficient. However, because inequality is not transitive, *three* inequalities on Cno are necessary. We also note that the use of DISTINCT is again necessary to eliminate duplicates.

An interesting variation here is to replace the three inequalities among the Cno's in the WHERE clause by the condition (X.Cno < Y.Cno) AND (Y.Cno < Take.Cno). First, since less-than comparisons are transitive, only two of them are necessary. Second, this also substantially reduces the number of potential duplicates in the answer.

We now have a general approach that gives us a family of Type 1 solutions for "at least K" questions for any K. For "at least 4 courses" we would use 4 copies of Take, 3 equalities on Sno, and either 6 inequalities or 3 less-than comparisons among Cno's. For "at least 5 courses" we would use 5 copies of Take, 4 equalities and either 10 inequalities or 4 less-thans, etc. (The formula for computing the number of required inequalities is K*(K-1)/2.)

Clearly, these solutions become very bulky and inefficient for all but the smallest values of K. We will show later how the questions involving "at least" can also be posed in a more concise and efficient way using aggregation. Nonetheless, the ability to pose them as Type 1 queries is important.

9 Negation Revisited

We now consider the remaining questions from our standard list: questions E8 through E13, as well as question E5. As we have already noted, question E5 involves real negation— the kind that cannot be phrased as a Type 1 query. Although not obvious at the first glance, these remaining questions also involve real negation— to see that, consider rephrasing them as shown in Figure 9.1. (Negation is italicized for emphasis.)

While the rephrasings for questions E12 and E13 are somewhat obscure (they are, nevertheless, correct), the rest of the rephrasings are quite intuitive. Thus, we have essentially reached a dead-end— to proceed further we need real negation, and Type 1 SQL cannot give it to us. Not to worry, however—that's why we have the rest of SQL.

E8. Who takes *at most* 2 courses?
is equivalent to
Who *does not* take at least 3 courses?

E9. Who takes *exactly* 2 courses?
is equivalent to
Who takes at least 2 courses and *does not* take at least 3 courses?

E10. Who takes *only* CS112?
is equivalent to
Who takes CS112 and *does not* take any other course?

E11. Who takes *either* CS112 *or* CS114?
is equivalent to
Who takes CS112 or CS114, and *does not* take both CS112 and CS114?

E12. Who are the *youngest* students?
is equivalent to
Who *are not among* those students who *are not* youngest?

E13. Who takes *every* course?
is equivalent to
Which students *are not among* those for whom there is a course that they *do not* take?

Figure 9.1: Questions E8 through E13 rephrased to explicitly show negation.

The positive aspect of this, however, is that once we master real negation (i.e., question E5) posing questions E8 through E11 becomes quite straightforward because they simply combine negation with the previously considered standard question types. Specifically,

E8. Construct "at most" is a combination of "does not" and "at least."

E9. Construct "exactly" is a combination of "does not" and two "at leasts."

E10. Construct "only" is a combination of "does not" and "is not" (or, "other than").

E11. Construct "either-or" (or "exclusive or," as it is conventionally called) is a combination of "or," "does not" and "both-and."

Questions E12 (construct "youngest") and E13 (construct "every") will be considered separately.

10 Type 2 SQL Queries

A sample Type 2 query is shown in Figure 10.1. We begin the discussion by first explaining its syntax, then introducing the Type 2 evaluation mechanism, and finally tracing this query to determine the question it poses.

```
SELECT Sno, Sname
FROM Student
WHERE (Sno IN
        (SELECT Sno
        FROM Take
        WHERE (Cno = "CS112")))
```

Figure 10.1: A sample Type 2 SQL query.

Syntactically, a Type 2 SQL query is a collection of several *non-correlated* component queries, with some of them nested in the WHERE clauses of the others. (The meaning of the term "non-correlated" will be explained in a moment.)

Theoretically, there is no limit on the number of nesting levels or on the number of component queries involved. On a practical level, however, SQL compilers do impose some limitations in this regard (e.g., no more than 16 nesting levels or 256 component queries), but these rarely present any real problems.

In the above example, we have two component queries. The inner query (called the *subquery*) is nested in the WHERE clause of the outer query (called the *main query*), and the queries are connected by the list-membership operator IN. (Out of the three pairs of parenthesis used in this example,

the only pair that is actually required is the left parenthesis immediately before SELECT and its matching right parenthesis.)

A query is called *non-correlated* if all column references in it are *local*— i.e., all columns come from, or *are bound to*, the tables in the local FROM clause. Any non-local column reference is called a *correlation*, and the query containing it becomes *correlated*.

In determining these column-to-table bindings, SQL follows the standard "inside-out, try the local scope first" scope rules. Specifically, given a column reference, SQL first tries to find some table in the local FROM clause containing that column. If the column reference also involves a prefix— either a table name or an alias— then SQL also looks for the match on that prefix. Three alternative outcomes are then possible.

1. SQL finds a *single such table* and successfully binds that column reference to that table.
2. SQL finds *several such tables*. Then column reference is *ambiguous* and the appropriate error message is generated.
3. SQL *does not find any such table* in the local FROM clause. Then column reference is *not local*. SQL then looks at the next outer scope— i.e., at the FROM clause in the immediately enclosing query, and attempts to bind that column reference to a table in that FROM clause.

This process continues (using the same three possibilities) until either a column reference is successfully bound, or an ambiguity is found, or the binding process "falls off" the main query, in which case the column reference cannot be bound at all, and the appropriate error is declared. (This

last case corresponds to an undeclared variable in conventional programming languages.)

Given this process and the query of Figure 10.1, columns Sno and Cno in its subquery are bound to the inner Take table, and columns Sno and Sname in its main query are bound to the outer Student table, thus making all bindings local, the component queries non-correlated, and this entire query of Type 2.

The formal syntactic condition for a multi-level (i.e., with subqueries) query to be of Type 2 is presented in Figure 10.2.

A query with subqueries is of Type 2 if every component query in it is non-correlated or, equivalently, if every column reference in it is local.

Figure 10.2: The definition of a Type 2 query.

The evaluation mechanism for Type 2 queries is presented in Figure 10.3.

To execute a Type 2 query, execute its component queries in the "inside-out" order— i.e., with the inner-most nested subquery first, replacing each query by its result as it gets evaluated.

Figure 10.3: The Type 2 SQL evaluation mechanism.

In case of the query of Figure 10.1, the subquery

```
SELECT Sno
FROM Take
WHERE (Cno = "CS112")
```

which is a regular Type 1 query, is executed first. Its answer is then substituted into its place in the main query

```
SELECT Sno, Sname
FROM Student
WHERE (Sno IN (...))
```

which is executed next. Note that, at this point, the main query has been reduced to a simple Type 1 query. Also note that, since the subquery retrieves a single column in its SELECT clause, the answer to it is just a list of values. Thus, the use of the list membership operator IN as the inter-query connector is quite appropriate.

We note that Type 2 queries fundamentally involve multiple data passes — one for each component query. In this case, the first pass is through table Take in the subquery, and the second pass is through table Student in the main query.

To determine the question posed by this query, observe that the subquery is a verbatim copy of query Q1, and thus poses question E1— *Who takes CS112?* The main query, which retrieves their student numbers and names then corresponds to the question *What are the student numbers and names of students who take CS112?* In other words, this is just another way of asking our standard question E2.

Before concluding this section, we note that, from a syntactic point of view, the condition (Sno IN ...) of the main query is just that— a condition; thus, it can itself be part of a more complex Boolean expression involving NOT, AND and OR— a feature that will become very handy in a moment.

11 Using Type 2 to Implement Real Negation

As we have discussed in Section 7, questions involving real negation need two passes through the data, using the following general strategy:

> *To pose a question "Who does not do X?"*
> 1. *identify and select those who actually do X; and*
> 2. *remove them from the list of those who potentially may do X.*

Since Type 2 queries fundamentally involve multiple data passes, they give us exactly what is necessary to implement real negation in SQL. Consider the query of Figure 11.1.

```
SELECT Sno
FROM Student
WHERE NOT (Sno IN
                (SELECT Sno
                 FROM Take
                 WHERE (Cno = "CS112")))
```

Figure 11.1: A mystery Type 2 SQL query involving negation.

This query differs from the query of Figure 10.1 in two ways: there is a NOT in front of the main query's condition, and Sno is used alone in the main SELECT clause. But how is this query evaluated, and what question does it pose?

Since this is a Type 2 query, it is evaluated inside-out. Again, the subquery is the same as query Q1 and poses question E1— *Who takes CS112?* Because of the NOT operator in its WHERE clause, the main query now retrieves the

student numbers of *all other students*— i.e., those student numbers that are not on the list generated by the subquery. Thus, the full query of Figure 11.1 corresponds to the question *Who does not take CS112?*— i.e., our standard question E5. (We will refer to this query as Q5.)

An interesting related query is shown in Figure 11.2.

```
SELECT Sno
FROM Take
WHERE NOT (Sno IN
                (SELECT Sno
                 FROM Take
                 WHERE (Cno = "CS112")))
```

Figure 11.2: Using Take in both FROM clauses.

This query was obtained from the query of Figure 11.1 by replacing table Student with table Take in the main FROM clause.

Note that this is still a Type 2 query, with the inner Sno and Cno coming from the inner Take table, and the outer Sno coming from the outer Take table. (Even though we use two copies of table Take here, because of the scope rules, no binding ambiguities arise and no aliases are necessary.)

The use of table Take in the main FROM clause limits the list of students who may potentially appear in the answer to whose who are listed in Take and thus take some course. Thus, it corresponds to the question *Who takes some (i.e., at least 1) course, but does not take CS112?*

As we can see, this query is different from the query of Figure 11.1, just as this question is different from our original question E5— *Who does not take CS112?*

12 Posing Questions Involving "at most," "exactly," "only," and "either-or"

Using the solution to question E5 as a foundation, it is now quite easy to generate solutions to questions E8 through E11. Specifically, question E8— *Who takes at most 2 courses?*— is posed by query Q8 of Figure 12.1.

```
SELECT Sno
FROM Student
WHERE NOT (Sno IN
            (SELECT X.Sno
             FROM Take X, Take Y, Take
             WHERE (X.Sno = Y.Sno)
                AND (Y.Sno = Take.Sno)
                AND (X.Cno != Y.Cno)
                AND (Y.Cno != Take.Cno)
                AND (X.Cno != Take.Cno)))
```

Figure 12.1: Query Q8 for question E8 – Who takes at most 2 courses?

This query is based on rephrasing question E8 as *Who does not take at least 3 courses?* The subquery here poses the question *Who takes at least 3 courses?* and is taken from Figure 8.2. The NOT in the main WHERE clause achieves the desired negation.

We note that this query will retrieve students who do not take any courses. This is appropriate, since "at most 2" means 0 (!), 1 or 2.

By using Take instead of Student in the main FROM clause we can change this query to ask a related question—*Who takes some (i.e., at least 1), but at most 2, courses?* In other words, *Who takes 1 or 2 courses?*

Question E9— *Who takes exactly 2 courses?*— is posed by query Q9 of Figure 12.2.

```
SELECT X.Sno
FROM Take X, Take
WHERE (X.Sno = Take.Sno)
   AND (X.Cno < Take.Cno)
     AND NOT  (X.Sno IN
                  (SELECT X.Sno
                   FROM Take X, Take Y, Take
                   WHERE (X.Sno = Y.Sno)
                      AND (Y.Sno = Take.Sno)
                      AND (X.Cno < Y.Cno)
                      AND (Y.Cno < Take.Cno)))
```

Figure 12.2: Query Q9 for question E9 – Who takes exactly 2 courses?

This query is based on rephrasing question E9 as *Who takes at least 2 courses and does not take at least 3 courses?* The main query poses *Who takes at least 2 courses?* and is copied from query Q7. (We have used the less-than operator in its condition to remove duplicates from the final answer.) The subquery again poses *Who takes at least 3 courses?* (Here, we have used less-than operators for conciseness.) The NOT in the main WHERE clause again achieves the desired negation.

We note that even though we have used the same alias name X in both the main query and in the subquery, because of the scope rules, no confusion arises.

Question E10— *Who takes only CS112?*— is posed by query Q10 of Figure 12.3.

```
SELECT Sno
FROM Take
WHERE (Cno = "CS112")
    AND NOT (Sno IN
                (SELECT Sno
                 FROM Take
                 WHERE (Cno != "CS112")))
```

Figure 12.3: Query Q10 for question E10 – Who takes only CS112?

This query is based on rephrasing question E9 as *Who takes CS112 and does not take any other course?* Here, the main query is based on query Q1— *Who takes CS112?* The subquery is taken verbatim from query Q6— *Who takes a course which is not CS112?* The NOT in the main WHERE clause again achieves the desired negation.

Two things should be noted about query Q10. First, thinking that the outer NOT can be brought inside the WHERE clause of the subquery, and thus cancel the negation implied by the != operator, is a common mistake— it cannot. Doing so would change the meaning of the query.

Second, the condition (Cno = "CS112") in the main WHERE clause is unnecessary *in this case* (!) and can be dropped *without changing the query's meaning.* The simplified query is shown in Figure 12.4. (Before proceeding further, try tracing it to determine why this is so.)

```
SELECT Sno
FROM Take
WHERE NOT (Sno IN
            (SELECT Sno
             FROM Take
             WHERE (Cno != "CS112")))
```

Figure 12.4: Query of Figure 12.3 simplified.

For a particular Sno to appear in the final result, it must not appear with any Cno value different from "CS112" in table Take. (If it did, the NOT ... IN of the main condition would "remove" that Sno from the answer.) But, for this Sno to even be considered in the main query, it must come from some Take row, and, thus must have some Cno associated with it in that row. Since this Cno cannot be anything other than CS112, it must be CS112. Thus, explicitly testing for it is unnecessary.

Finally, question E11— *Who takes either CS112 or CS114?*— is posed by query Q11 of Figure 12.5.

```
SELECT Sno
FROM Take
WHERE ((Cno = "CS112") OR (Cno = "CS114"))
   AND NOT (Sno IN
             (SELECT X.Sno
              FROM Take X, Take
              WHERE  (X.Sno = Take.Sno)
                  AND (X.Cno = "CS112")
                  AND (Take.Cno = "CS114")))
```

Figure 12.5: Query Q11 for question E11 – Who takes either CS112 or CS114?

This query which is based on rephrasing question E11 as *Who takes CS112 or CS114, and does not take both CS112 and CS114?* is a combination of query Q3— *Who takes CS112 or CS114?*— for the main query, and query Q4— *Who takes both CS112 and CS114?*— for the subquery, with the outer NOT achieving the negation.

Note the extra pair of parenthesis around the OR in the main condition. This forces the OR to be executed before the AND in the main condition— otherwise, the SQL standard *rules of precedence* for logical operators are: first NOT, then AND, and only lastly OR.

13 Computing Extremes as Type 2 Queries with Negation

Consider question E12— *Who are the youngest students?* While the most natural way of posing this question involves the use of the aggregate function MIN() (to be presented later), this question can also be posed using a Type 2 query with negation, as shown in Figure 13.1.

```
SELECT Sno
FROM Student
WHERE NOT (Age IN
            (SELECT X.Age
            FROM Student X, Student
            WHERE (X.Age > Student.Age)))
```

Figure 13.1: Query Q12 for question E12 – Who are the youngest students?

This solution is interesting because it looks at the problem of minimums (and maximums) in a novel way. It is also important, because, as we will point out later, SQL imposes certain limitations of the use of its aggregate functions. Thus, ability to ask this and similar questions without the use of aggregates is important.

To see how this query operates, let's trace its execution. (You should follow this trace on paper.) Since this is a Type 2 query, it is evaluated inside-out. That means that the subquery is evaluated first. But, what does this subquery retrieve?

It retrieves a particular Age value if there is some other Age value less than it. In other words, it retrieves those Age values which are *not smallest.* To rephrase this even shorter, it retrieves *all* Ages *except the smallest one.*

Given each student, the main query then tests whether his age is not in the list of ages retrieved by the subquery. Of course, since the only age not present in the list is the smallest one, only the students possessing it— i.e., the youngest ones— will be retrieved in the final answer.

If all students happen to be of the same age, then all students would be youngest. In that case, the answer to the subquery would be *empty;* the main WHERE clause would be True for all Student rows; and all students would be retrieved.

Note that, in Figure 13.1, the connection between the main query and the subquery is made on the Age column. However, it is also possible to ask the same question by making this connection on the Sno column, by replacing the WHERE clause as follows. (The choice between these two alternatives is a matter of personal preference.)

```
WHERE NOT (Sno IN
                (SELECT X.Sno
                FROM Student X, Student
                WHERE (X.Age > Student.Age)))
```

If we wish to pose the question *Who are the oldest students?,* we can simply replace the greater-than operator in the WHERE clause of the subquery with the less-than operator.

14 A Look at SQL Enhancements

In this section, we take a break from the fundamental features of the language and take a brief look at the various enhancements it provides. Roughly speaking, these enhancements can be divided into three categories: *syntactic enhancements, presentation enhancements* and *scalar expressions.*

Syntactic enhancements are those features that make the *writing* of SQL queries easier. They include: omitting the WHERE clause, using symbol * (asterisk) in the SELECT clause, and using operator NOT IN.

SQL permits the omission of the WHERE clause if the condition of the query is always True— i.e., there really is no condition that needs to be imposed. Thus, a request to retrieve the student numbers of *all* students is posed by the query of Figure 14.1.

```
SELECT Sno
FROM Student
```

Figure 14.1: Example of query with the omitted WHERE clause.

SQL provides the shorthand symbol * (asterisk) if the request is to retrieve *all* columns from some table in the FROM clause. For example, the query of Figure 14.2 retrieves *all information* about students who take CS112.

```
SELECT Student.*
FROM Student, Take
WHERE (Student.Sno = Take.Sno)
   AND (Cno = "CS112")
```

Figure 14.2: Using symbol * to mean "all columns."

We note that using symbol * without the prefix, as in

```
SELECT *
```

causes *all* columns from *all* tables in the FROM clause to be retrieved.

The NOT IN operator is a syntactic enhancement that allows NOT to be brought inside parenthesis containing IN. In other words, any construct of the form

```
NOT (<something> IN (SELECT... FROM... WHERE... ))
```

can be equivalently rewritten (i.e., without changing its meaning) as

```
(<something> NOT IN (SELECT... FROM... WHERE... ))
```

While NOT IN is actually a primitive operator in SQL, we can think of it simply as a shorthand for the longer NOT ... IN combination, making such combinations easier to read.

Presentation enhancements make answers easier to understand. They include: suppressing duplicates in the answer, assigning new names to columns of the answer, and sorting the answer.

As previously discussed, using the keyword DISTINCT, we can suppress duplicates from the answer.

We emphasize that DISTINCT eliminates *duplicate rows*, and not individual values, from the answer. A simple query that makes this point clear is shown in Figure 14.3.

```
SELECT DISTINCT Sname, Age
FROM Student
```

Figure 14.3: Suppressing duplicates from the answer.

This query would eliminate an answer row only if it duplicates *both* the Sname value and the Age value from some other answer row. Thus, it would leave both rows <"John Doe", 20> and <"John Doe", 23> in the answer. (Recall that Sno, and not Sname, is the key for table Student; so, there can be two students, each named John Doe.)

Often, names of columns in the SELECT clause are not the ones desired for the column headings in the answer. To overcome this, SQL allows for the assignment of new column headings. Depending on the SQL dialect, there are basically two ways to do this. We can prefix the column reference in the SELECT clause with its new name followed by the equal sign.

```
SELECT Name = Sname, ...
```

Alternatively, we can follow the column reference by the keyword AS and the new name.

```
SELECT Sname AS Name, ...
```

In both cases, when we want the new column heading to include spaces (and, in some SQL dialects, certain other characters as well), the new name should be enclosed in quotes.

```
SELECT "Student Name" = Sname, ...
```

The sorting of a query answer can involve one or several columns from the answer, and may also specify the direction (ascending or descending) for each of these columns. As an example, consider the query of Figure 14.4.

```
SELECT Sname, Age
FROM Student, Take
WHERE  (Student.Sno = Take.Sno)
    AND (Cno = "CS112")
ORDER BY Sname, Age DESC
```

Figure 14.4: Sorting facilities of SQL.

The sort specification is contained in the ORDER BY clause, which syntactically appears as the very last clause of the query, and which is executed after the SELECT clause.

The argument to the ORDER BY clause is a list of columns from the SELECT clause, where the left-to-right column order defines the major-to-minor sort sequence. Each column can optionally be followed by the keywords ASC or DESC, defining the sort order on that column as ascending or descending, respectively, with ASC as the default if omitted.

In this case, the answer would first be sorted by Sname ascending, and then, within each group of rows with the same Sname, by Age descending.

An interesting feature of the ORDER BY clause is that it also allows us to refer to columns from the SELECT clause by their position. Furthermore, mixing of column names and positions is allowed. Thus, for example, the ORDER BY clause from the query of Figure 14.4 can be equivalently rewritten as

```
ORDER BY Sname, 2 DESC
```

Notably, this is the *only place* in the entire SQL language where columns can be referenced by their positions. (This feature will become useful in a moment.)

We note that changing the ORDER BY clause in our query to

```
ORDER BY Age DESC, Sname
```

only changes how the answer is displayed row-wise— it will now be sorted first by Age descending, and then by Sname. Columns will still be displayed as specified in the SELECT clause— with Sname column on the left, and Age column on the right.

Finally, the behavior of sorts is always consistent with the behavior of the less-than and greater-than operators. For example, if the less-than operator treats strings as left-justified, the ascending sort would place them in the standard *lexicographic* (dictionary) ordering.

The most important enhancement feature of SQL is the availability of scalar expressions. Luckily, their use is very intuitive. Basically, SQL provides a reasonable set of scalar operators and functions for operating on numerics, strings and other data types, and allows their use in all reasonable places in the SELECT and WHERE clauses of queries.

While specifics as to which operators and functions are actually provided differ widely among SQL dialects, as an example here is a sampling of what Transact SQL provides. For numerics, there are the four basic arithmetic operators (+, -, * and /), modulo division (%), functions round(), trunc() and abs(), exponentiation function power(), logarithmic functions log() and log10() for natural and base-10 logarithms, a full set of trigonometric functions, etc.

For strings, there is a concatenation operator (+) and a set of substring functions, etc.

Other data types (most notably, *datetime*— for dealing with calendar dates and time values) have rich sets of functions as well.

The basic rule of thumb for using scalar expressions in SQL is as follows: Wherever one can use a column name in the SELECT or WHERE clauses, one can also use an expression of an appropriate data type comprised of column names, constants and various scalar operators and functions.

For example, the query of Figure 14.5 assumes table Room(Rno, Length, Width) and retrieves room numbers and areas of those rooms where the length of the room is within 1% of its width, sorting the answer in the descending order by area.

```
SELECT Rno, Area = Length*Width
FROM Room
WHERE abs(Length-Width)/Width <= 0.01
ORDER BY 2 DESC
```

Figure 14.5: Example use of arithmetic in SQL.

Since many SQL dialects forbid the use of expressions in the ORDER BY clause, and since in most of them column renaming is done after the sorting, referring to expression-based columns by their positions in the ORDER BY clause is the only means left to achieve the desired sort.

A less obvious consequence of having expressions in SQL, is that we can use the string manipulation capabilities for formatting query answers. As an example, Figure 14.6 shows a query that will retrieve all professor names, formatted as last name, comma, space, first initial, period, as in: "Smith, J." (Function substring(Fname,1,1) gets one character starting at position 1 from Fname; when used with strings, symbol + denotes the concatenation operator.)

```
SELECT Lname + ", " + substring(Fname,1,1) + "."
FROM Professor
```

Figure 14.6: Formatting an answer.

15 Handling Composite Keys

All of the questions considered thus far in this essay have been about students taking courses. Surely, the same types of questions can also be asked about professors teaching them. Are there any differences, and do the same types of solutions apply?

The answer is: Yes, the same types of solutions apply, but with an adjustment. Specifically, since table Professor has a composite, two column— Fname and Lname— key, any professor-based connection (positive or negative) between tables Professor and Teach, or between multiple copies of Professor or of Teach, has to involve *both* of these columns.

To illustrate this point, we briefly go through some of our standard questions, restated with respect to professors teaching courses. We begin with questions not involving negation.

The query of Figure 15.1 poses the question *What are full names and ages of professors who teach CS112?*, which is similar to our standard question E2.

```
SELECT P.Fname, P.Lname, Age
FROM Professor P, Teach T
WHERE  (P.Fname = T.Fname)
    AND (P.Lname = T.Lname)
    AND (Cno = "CS112")
```

Figure 15.1: Query for the question What are full names and ages of professors who teach CS112?

Questions about professors teaching courses involving "or," "both-and," and "is not" (similar to standard questions E3, E4, E6) are posed in a similar fashion, and are left as an exercise for the reader.

The question *Who teaches at least 2 courses?* (similar to question E7) is also quite straightforward, and is posed in a manner similar to query Q7.

Another question also involving "at least"— *Which courses are taught by at least 2 professors?*— but formulated with respect to courses taught by professors, however, takes more thought. (Try posing it before proceeding.)

The tricky part here is to realize that a difference in just the first names or just the last names of two professors is sufficient to make them different from each other. The correct way of expressing this condition involves the use of OR, as shown in Figure 15.2, with the necessary extra pair of parenthesis around the OR operands. As an additional exercise, try posing the question *Which courses are taught by at least 3 professors?*

```
SELECT X.Cno
FROM Teach X, Teach Y
WHERE   (X.Cno = Y.Cno)
    AND ((X.Fname != Y.Fname)
      OR  (X.Lname != Y.Lname))
```

Figure 15.2: Query for the question Which courses are taught by at least 2 professors?

16 Negation Over Composite Keys

Posing negative questions regarding professors teaching courses presents its own set of interesting problems. Consider the question *Who does not teach CS112?* — similar to our question E5— along with the query of Figure 16.1.

```
SELECT Fname, Lname
FROM Professor
WHERE (Fname NOT IN
        (SELECT Fname
        FROM Teach
        WHERE (Cno = "CS112")))
AND     (Lname NOT IN
        (SELECT Lname
        FROM Teach
        WHERE (Cno = "CS112")))
```

Figure 16.1: First attempt at the question Who does not teach CS112?

However intuitive, this query is nonetheless incorrect for this question. (Try to determine why before proceeding.)

The problem here is that the lists of the first names and the last names of professors who do teach CS112 are compiled and tested *independently* of each other. Thus, they are not "synchronized" to come from the same Teach row. As a result, professors who do not in fact teach CS112 may incorrectly be suppressed by this query.

To see this, consider an example where table Professor contains only two rows, and table Teach contains only a single

row, as shown in Figure 16.2. (Since the Dept, Rank, Salary and Age values are immaterial for this example, we are not showing these columns in table Professor.)

Professor		
Fname	**Lname**	...
"John"	"Smith"	
"Mary"	"Smith"	

Teach		
Fname	**Lname**	**Cno**
"John"	"Smith"	"CS112"

Figure 16.2: Example data for tables Professor and Teach.

A trace of the query of Figure 16.1 on this data shows that it will return an empty result. Yet, Mary Smith does not teach CS112, and thus should be in the answer to the question *Who does not teach CS112?*

Changing this query to use OR in place of AND does not fix the problem either. As an example here, consider tables Professor and Teach as shown in Figure 16.3.

Professor		
Fname	**Lname**	...
"John"	"Smith"	
"Mary"	"Smith"	
"John"	"Brown"	
"Mary"	"Brown"	

Teach		
Fname	Lname	Cno
"John"	"Smith"	"CS112"
"Mary"	"Smith"	"CS112"
"John"	"Brown"	"CS112"

Figure 16.3: Another example data for tables Professor and Teach.

The answer to such a modified query on the data of Figure 16.3 would again be empty. Yet, the answer to our question here should list Mary Brown.

Note that we seem to have run out of available syntax at this point— there does not seem to be any other reasonable query modification. Does that mean that we cannot ask such a question? The answer is: Of course, we can— we just need to recall and appropriately use one of the previously introduced language features.

We must first determine what it is that we actually need. The concept of a composite key offers a clue: We do not want first names and last names to be treated separately. Rather, we want NOT IN to be tested between *Fname/Lname pairs.* In other words, we would like to be able to write a query as shown in Figure 16.4.

```
SELECT Fname, Lname
FROM Professor
WHERE (Fname, Lname NOT IN
        (SELECT Fname, Lname
          FROM Teach
          WHERE (Cno = "CS112")))
```

Figure 16.4: Desired query form for the question Who does not teach CS112?

Unfortunately, this is syntactically incorrect: the IN and NOT IN operators cannot be used with a list of "pairs." They can, however, be used with a list of *expression results!*

In other words, even though the IN and NOT IN operators require that the subquery retrieve a single column in its SELECT clause, the specification for that column need not be a single basic attribute, but can be an expression. Likewise, the object to the left of IN and NOT IN can also be an expression. All that is required is that the data types of these expressions be compatible.

The query of Figure 16.5 incorporates expressions to pose our question *Who does not teach CS112?*

```
SELECT Fname, Lname
FROM Professor
WHERE (Fname + Lname NOT IN
        (SELECT Fname + Lname
         FROM Teach
         WHERE (Cno = "CS112")))
```

Figure 16.5: Using string concatenation expressions for the question Who does not teach CS112?

We assume here that columns Fname and Lname are implemented as fixed character strings, and that the concatenation operator + does not suppress extra spaces, so that Fname and Lname do not "intrude" into each other.

Using expressions in this manner, we can now pose questions involving "at most," "exactly," "only" and "either-or" (similar to questions E8 through E11) about professors teaching courses, as well as a question involving "youngest" (or "oldest") (similar to question E12) about professors. These are left as exercises for the reader.

17 Standard Questions Involving "every"

Given an ability to use expressions around the IN and NOT IN operators, we can now pose question E13— *Who takes every course?*— as shown in Figure 17.1.

```
SELECT Sno
FROM Student
WHERE (Sno NOT IN
        (SELECT Sno
        FROM Student, Course
        WHERE  (Sno + Cno NOT IN
                (SELECT Sno + Cno
                FROM Take))))
```

Figure 17.1: Query Q13 for question E13 – Who takes every course?

This query, which is motivated by the rephrasing of question Q13 as *Which students are not among those for whom there is a course that they do not take?* (see Figure 9.1) is quite complex, and really has to be traced in order to understand how it works.

Because all column references in this query are local, this query is of Type 2. (The inner Sno and Cno come from the inner table Take, the middle Sno and Cno come from the middle tables Student and Course respectively, and the outer Sno comes from the outer table Student.) Therefore, it is executed inside-out.

We first evaluate the inner-most subquery, which returns a list of Sno/Cno concatenations from table Take.

We then evaluate the middle subquery. This subquery executes a cross-product between tables Student and Course. Its NOT IN operator then retains only those wide rows from this cross-product where the Sno/Cno combinations are *not retrieved* by the inner subquery— i.e., *not present* in table Take— i.e., where student Sno *does not take* course Cno. The answer to the middle subquery then contains the Sno values from these combinations— i.e., the student numbers of those students for whom there is a course that they do not take.

The condition of the outer query then fails for these students, leaving for the final answer the student numbers of only those students who *are not among* those retrieved by the middle subquery. In other words, those students who take every course.

18 A Brief Interlude

We have just completed all of the standard questions from Figure 4.1, covering Type 1 and Type 2 queries, as well as SQL enhancement features. What remains are: the aggregation facilities, NULLs (remember the NULL and NOT NULL qualifiers from Figure 3.1?), the Type 3 queries, the DML component of SQL— namely, the facilities to insert, delete and update rows, and SQL extensions.

19 Additional Standard Questions

Five additional standard questions, motivating SQL aggregation facilities, are listed in Figure 19.1.

E14.	For each department that has more than 3 professors older than 50, what is the average salary of such professors? (*A related question:* For each department/rank combination that has more than 3 professors older than 50, what is the average salary of such professors?)
E15.	What is the grade point average (GPA) of each student?
E16.	What is the overall average salary of all professors who are older than 50?
E17.	Whose (i.e., which professors') salary is greater than the overall average salary?
E18.	Whose salary is greater than the average salary within that professor's department?

Figure 19.1: A list of additional standard questions motivating SQL aggregation facilities.

While the standard nature of these questions is not immediately apparent, it will become clear as we consider each question in turn.

20 Computing Aggregates for Groups

The query posing question E14— *For each department that has more than 3 professors older than 50, what is the average salary of such professors?*— is shown in Figure 20.1.

```
SELECT Dept, AVG(Salary)
FROM Professor
WHERE (Age > 50)
GROUP BY Dept
HAVING (COUNT(*) > 3)
```

Figure 20.1: Query Q14 for question E14 – For each department that has more than 3 professors older than 50, what is the average salary of such professors?

This query involves two new clauses: the GROUP BY clause, which takes a list of columns as an argument, and the HAVING clause, which takes a condition as an argument. It also involves two *aggregate functions:* AVG() and COUNT(*).

Syntactically, the HAVING clause is positioned immediately after the GROUP BY clause. The two are placed after the WHERE clause, but before the ORDER BY clause (not present in this case).

Operationally, the following sequence of events takes place. First, SQL formulates and evaluates the query

```
SELECT *
FROM Professor
WHERE (Age > 50)
```

This query is called the *underlying query*. It is formed from our original query by dropping the GROUP BY and HAVING clauses and by replacing the SELECT specification with symbol * (asterisk). Thus, it retrieves *all* columns from *all* of the tables involved. (While, in this case, the underlying query uses only one table and is of Type 1, in general it can involve multiple tables and be of any type.)

The result of this underlying query is then *grouped by* the Dept column (or, more precisely, by the values remaining in this column). In other words, the rows are re-arranged (or, re-ordered) in such a way that all rows sharing the same Dept value are listed next to each other.

We note that grouping is not the same as sorting; in particular, grouping does not guarantee that among themselves the groups would be listed in any particular order. Thus, for example, a group of "Mathematics" rows may appear after the group of "English" rows, but before the group of "History" ones.

SQL then evaluates the HAVING clause condition (COUNT(*) > 3) for each group, and eliminates those groups for which this condition fails. When evaluated for a group, the aggregate function COUNT(*) returns the number of rows in that group. Thus, for our query, only those groups that have more than 3 rows in them remain.

Finally, SQL formulates one resulting row corresponding to each remaining group and retrieves it. In this case, according to the SELECT clause of our query

```
SELECT Dept, AVG(Salary)
```

this resulting row would contain the Dept value of the group, and the average of all its Salary values.

This overall evaluation process involves four steps, as listed in the evaluation mechanism shown in Figure 20.2.

1. Formulate and evaluate the appropriate underlying query

```
SELECT *
FROM ...
WHERE ...
```

2. Group (re-arrange) the rows in the result of Step 1 according to the GROUP BY clause.
3. Evaluate the HAVING clause condition for each group, and eliminate those groups for which this condition fails.
4. Formulate one resulting row for each remaining group, according to the SELECT clause, and retrieve it.

Figure 20.2: The evaluation mechanism for queries involving aggregation.

We emphasize that this is a formal evaluation mechanism. As such it may differ from, and in particular appear less efficient than, the actual evaluation strategies taken by real SQL systems. However, we do guarantee that it will always generate the same result.

In addition to COUNT(*) and AVG(), SQL provides four other aggregate functions: MIN(), MAX(), SUM(), and COUNT(DISTINCT <column>), that also can be used in the SELECT and HAVING clauses. The HAVING clause can also involve Boolean operators NOT, AND and OR, as well as scalar expressions. No subqueries are allowed in the HAVING clause, however! These capabilities are illustrated by the query of Figure 20.3

```
SELECT Dept,
        MIN(Salary), MAX(Salary), AVG(Salary),
        COUNT(DISTINCT Salary)
FROM Professor
WHERE (Age > 50)
GROUP BY Dept
HAVING (COUNT(*) > 3)
     OR (SUM(Salary) - 200000  > 0)
```

Figure 20.3: A query showing the use of all aggregate functions, and also showing the use of Boolean expressions in the HAVING clause.

For numeric arguments, the behavior of MIN(), MAX(), AVG() and SUM() is quite natural: given a column of values, they return the *smallest* value, the *largest* value, the *average* of the values, and the *arithmetic sum* of the values, respectively. (We note that the aggregate function AVG() *does not* suppress duplicate values in its argument.)

For non-numeric data types, functions AVG() and SUM() are generally not defined, and the behavior of MIN() and MAX() is consistent with the behavior of the less-than and greater-than operators for those data types. For example, function MIN() applied to a column of string values, would return the lexicographically smallest of them. (Recall that the sorting behavior imposed by the ORDER BY clause is also consistent with the behavior of the less-than and greater-than operators.)

Functions COUNT(*) and COUNT(DISTINCT <column>) are both counters, and apply to all data types. *What* they count, however, is quite different. As we have already noted, function COUNT(*) simply counts the number of rows in a group.

Function COUNT(DISTINCT <column>) counts how many *different* <column>-values are present in the group. For example, given a group with 5 rows containing Salary values (20000, 30000, 20000, 20000, 30000), function COUNT(DISTINCT Salary) would return 2— for the two distinct values: 20000 and 30000. (While this use of DISTINCT serves a different purpose than its use in SELECT DISTINCT, both uses are quite consistent with each other.)

The question corresponding to the query of Figure 20.3 can be phrased as follows: *For each department that has more than 3 professors older than 50 or where the total salary of such professors exceeds $200,000, what is the minimum salary, the maximum salary, the average salary, and the number of distinct salary values, of such professors?*

SQL permits the omission of the HAVING clause if the condition on the groups is always True— i.e., all groups are desired for the answer. (Recall the similar ability with the WHERE clause.) This allows the question *For each department, what is the average salary of those professors who are older than 50?* to be posed as shown in Figure 20.4.

```
SELECT Dept, AVG(Salary)
FROM Professor
WHERE (Age > 50)
GROUP BY Dept
```

Figure 20.4: The query for the question For each department, what is the average salary of those professors who are older than 50?

If the argument to the GROUP BY clause is a list of several columns, then the grouping occurs for all of them simulta-

neously. To illustrate this, consider the query of Figure 20.5, which poses the question *For each department/rank combination that has more than 3 professors older than 50, what is the average salary of such professors?* (the related question to E14).

```
SELECT Dept, Rank, AVG(Salary)
FROM Professor
WHERE (Age > 50)
GROUP BY Dept, Rank
HAVING (COUNT(*) > 3)
```

Figure 20.5: SQL query for the question For each department/ rank combination that has more than 3 professors older than 50, what is the average salary of such professors?

In this query, rows are grouped *both* by Dept and by Rank—in other words, all rows in a group must now agree both on Dept and on Rank values. (Generally, this creates more groups, each with fewer rows.) Since grouping does not imply sorting, the order of columns in the GROUP BY clause has no significance.

We note that in this case, the condition *"... that has more than 3 professors older than 50 ..."* will be applied to each department/rank combination separately. Thus, it is possible for some department to survive this condition in query Q14, but to *not survive* this condition here, when the department is further subdivided by rank. (Find such a data example, and trace the two queries.)

Standard SQL imposes an extremely important syntactic restriction on the form of the SELECT and HAVING clauses in queries involving aggregate functions and/or the GROUP BY clause:

Only those columns explicitly listed in the GROUP BY clause may appear un-aggregated (i.e., not as a parameter of an aggregate function) in the SELECT or HAVING clauses.

or, for those who prefer negative formulations:

It is prohibited to use an un-aggregated column in the SELECT or HAVING clauses, unless it is explicitly listed in the GROUP BY clause.

The motivation for this restriction is as follows. Since SQL generates a *single result row* for each group, the values making this row up must be *representative* — or, *deterministic*— of that group. Likewise, when SQL tests the HAVING clause condition for the group, the arguments involved in this test must also be representative of it.

The only columns for which this "representativeness" can be syntactically guaranteed are those found in the GROUP BY clause— by the very nature of grouping, all values for such a column will be the same within a group. Thus, these columns can appear un-aggregated in the SELECT or HAVING clauses. Reference to any other column in the SELECT clause and in the HAVING clause must be aggregated. As we will see later in Section 24, this syntactic restriction has some awkward consequences.

However, this restriction does not imply that all of the GROUP BY columns *must* appear in the SELECT or HAVING clauses. For example, modifying the SELECT clause of the query of Figure 20.5 to

```
SELECT Dept, AVG(Salary)
```

is perfectly legitimate. It would still retrieve one average salary for each department/rank group— we just would not know which Rank value would go with which average. Modifying it to

SELECT AVG(Salary)

would also be all right— except now we would not know either the department or the rank that goes with the averages.

Further, the restriction does not require any aggregates to be actually included in the SELECT or HAVING clauses. Thus, for example, modifying the SELECT clause of the query of Figure 20.5 to

SELECT Dept, Rank

is again syntactically correct.

In concluding this section, we note that while the standard nature of question E14 is not immediately apparent, what it does is compute some column aggregate(s)— in this case, AVG(Salary), for groups of rows organized by some other column(s)— Dept, with some condition imposed on the groups— (COUNT(*) > 3). We also note that questions involving computation of aggregates for groups are often phrased using the English word "by"— e.g., *Compute average salary **by** department*— and are thus easy to recognize.

21 Combining Scalar Expressions with Aggregation

One very useful property of SQL is its ability to combine scalar expressions with aggregation. As an example of one such combination in the SELECT clause, consider question E15— *What is the grade point average (GPA) of each student?*— posed by the query of Figure 21.1.

```
SELECT Sno,
       GPA=round(SUM(Grade*Credits)/SUM(Credits),2)
FROM Take, Course
WHERE (Take.Cno = Course.Cno)
GROUP BY Sno
```

Figure 21.1: Query Q15 for question E15 – What is the grade point average (GPA) of each student?

Here, the specification for the second column of the SELECT clause is an expression consisting of two occurrences of the aggregate function SUM() and three scalar operations: multiplication, division and the scalar function round().

Because the multiplication (Grade*Credits) appears *inside* the argument to the aggregate function SUM(), it is carried out on the *individual* Grade and Credits values, *before* aggregation. Conversely, the division appears *outside* the SUM() aggregates and is performed *after* the aggregations on the *results* of the aggregates. The function round() is applied to the result of the division (occurring, therefore, after the aggregations) and, in this case, rounds it to 2 decimal digits.

Scalar expressions can be combined with aggregate functions in the HAVING clause as well. As an example, try extending query Q15 by adding a restriction that only GPAs over 2.5 should be retrieved.

We note that the ability to use scalar expressions as arguments to aggregate functions is a direct consequence of SQL's ability to substitute expressions for column names in the SELECT clause, and now in the HAVING clause as well. The only exception to this is with the COUNT(DISTINCT <column>) construct, where some SQL dialects prohibit the use of expressions for the <column> specification.

22 Some Old Questions Revisited

The questions involving "at least" posed in Section 8 as Type 1 queries, along with those involving "at most" and "exactly" posed in Section 12 as Type 2 queries, can also be posed very naturally using the queries with the GROUP BY and HAVING clauses. As an example, the question *Who takes at least 3 courses?* is posed by the query of Figure 22.1.

```
SELECT Sno
FROM Take
GROUP BY Sno
HAVING (COUNT(*) >= 3)
```

Figure 22.1: A query with the GROUP BY and HAVING clauses for the question Who takes at least 3 courses?

By replacing the condition (COUNT(*) >= 3) in the HAVING clause with the condition (COUNT(*) = 3), we get the question *Who takes exactly 3 courses?* Interestingly, by replacing it with (COUNT(*) <= 3) we get the question *Who takes at least 1, and at most 3, courses?* and not *Who takes at most 3 courses?* This is because the query uses table Take in the FROM clause, and thus students who take no courses at all are simply not considered by this query.

We note, however, that the ability to pose such questions using the GROUP BY and HAVING clauses does not mean that we can now forget their Type 1 and Type 2 implementations. As we will discuss later, SQL imposes certain restrictions on the use of GROUP BY and HAVING, and sometimes these questions have to be posed without them.

23 Global Aggregations

In this section, we look at a special case of aggregate queries— the ones dealing with "global" aggregations. As an example, consider question E16— *What is the overall average salary of all professors who are older than 50?*— posed by the query of Figure 23.1.

```
SELECT AVG(Salary)
FROM Professor
WHERE (Age > 50)
```

Figure 23.1: Query Q16 for question E16 – What is the overall average salary of all professors who are older than 50?

What's special about this query is the absence of the GROUP BY clause. With this syntactic form, the entire result of the underlying query, which in this case is

```
SELECT *
FROM Professor
WHERE (Age > 50)
```

is implicitly considered to be *a single group* for aggregation purposes. Thus, AVG(Salary) is computed *once* over *all* professors who are older than 50, generating a *single row* answer.

In queries involving global aggregations, SQL does not permit the mixing of the aggregated and un-aggregated expressions in the SELECT clause. This is because the presence of un-aggregated columns would violate the restriction from

Section 20— namely, that in queries involving aggregation, all un-aggregated columns in the SELECT clause must come from the GROUP BY clause.

Further, no condition can be imposed on the single underlying group in such queries. Such a condition must be placed in the HAVING clause, which in turn requires the explicit GROUP BY clause.

24 Comparing Base Values to Aggregates

Questions E17— *Whose (i.e., which professors') salary is greater than the overall average salary?*, and E18— *Whose salary is greater than the average salary within that professor's department?* represent an interesting category of standard questions which compare base values with aggregates.

Question E17 is posed by the query of Figure 24.1.

```
SELECT X.Fname, X.Lname
FROM Professor X, Professor Y
GROUP BY X.Fname, X.Lname, X.Salary
HAVING (X.Salary > AVG(Y.Salary))
```

Figure 24.1: Query Q17 for question E17 – Whose (i.e., which professors') salary is greater than the overall average salary?

We evaluate this query according to its evaluation mechanism.

1. We form and evaluate the underlying query

   ```
   SELECT *
   FROM Professor X, Professor Y
   ```

   This query involves a cross-product between two copies— X and Y— of table Professor. Because of the unqualified * (asterisk) in its SELECT clause, this query retrieves all columns from both copies of Professor.
2. We group the wide rows from the result of this cross-product by the combination of

X.Fname/X.Lname/X.Salary values from the X-copy of Professor. Each group then contains a set of wide rows, repeating the *same row* from the X-copy of Professor together with *all different rows* from the Y-copy. This, in effect, provides an entire Y-copy of table Professor for each Professor X-row.

3. We evaluate the condition (X.Salary > AVG(Y.Salary)) for each group, and eliminate those groups for which this condition fails. X.Salary here is the salary of that professor from the X-copy whose group is being considered. AVG(Y.Salary) computes the average salary of all professors from the Y-copy of Professor.
4. We then retrieve the X.Fname and X.Lname values from each surviving group.

The inclusion of X.Salary into the GROUP BY clause may seem unnecessary. As one might argue, since Fname and Lname together form the key for table Professor, and since they are already included in the GROUP BY, all Professor X-rows in a group will definitely share the same value of X.Salary anyway.

Recall, however, that there is an SQL *syntax rule* (see Section 20) that *requires* this inclusion for X.Salary to be used un-aggregated in the HAVING clause, regardless of any key, or other semantic, considerations.

An interesting variation of this query is shown in Figure 24.2. Here, because we have not included X.Salary in the GROUP BY clause, by the same rule, we must aggregate it in the HAVING clause. Note that, while computing the *smallest value* from a set of *equal values* might seem somewhat strange, this is precisely how we can satisfy SQL syn-

tax and at the same time get the X.Salary value for the group. (Functions MAX(Salary) and AVG(Salary) could also have been used here.)

```
SELECT X.Fname, X.Lname
FROM Professor X, Professor Y
GROUP BY X.Fname, X.Lname
HAVING MIN(X.Salary) > AVG(Y.Salary)
```

Figure 24.2: Alternative query form for question E17 – Whose (i.e., which professors') salary is greater than the overall average salary?

Question E18 is posed by the query of Figure 24.3.

```
SELECT X.Fname, X.Lname
FROM Professor X, Professor Y
WHERE (X.Dept = Y.Dept)
GROUP BY X.Fname, X.Lname, X.Salary
HAVING (X.Salary > AVG(Y.Salary))
```

Figure 24.3: Query Q18 for question E18 – Whose salary is greater than the average salary within that professor's department?

This query is very similar to query Q17. The addition of the WHERE clause condition (X.Dept = Y.Dept) assures that a professor's salary will be compared with the average of salaries from just his department.

25 Combining Type 2 Queries with Aggregation

As we have noted earlier, queries underlying aggregation can be of any type. As an illustration of the Type 2 underlying query, try posing the question *What is the grade point average (GPA) of each student, who does not take CS112?* (A closely related query will be shown in the next section.) Type 3 underlying queries will be discussed later.

SQL also allows for aggregates to be used in subqueries within Type 2 queries. As a motivating example, consider again question E12— *Who are the youngest students?*— along with the query of Figure 25.1.

```
SELECT Sno
FROM Student
WHERE (Age IN
        (SELECT MIN(Age)
         FROM Student))
```

Figure 25.1: Alternative query for question E12 — Who are the youngest students? — showing the use of aggregates inside the subqueries.

This query is a Type 2 query with the inner query that involves aggregation. Its trace shows that it correctly implements question E12— *Who are the youngest students?*

There is an interesting feature related to such queries. If the subquery involves global aggregation (i.e., if it involves aggregation, but does not have the GROUP BY clause), then

it can be syntactically guaranteed to retrieve a *single row.* We can then equivalently replace the operator IN with an equal sign. This, in turn, improves the readability of the WHERE clause, which in the case of the query of Figure 25.1 becomes

```
WHERE (Age =
        (SELECT MIN(Age)
        FROM Student))
```

All other binary comparators— !=, <, etc.— are also allowed in such cases. For example, our question E17— *Whose salary is greater than the overall average salary?*— can also be posed by the query of Figure 25.2.

```
SELECT Fname, Lname
FROM Professor
WHERE (Salary >
        (SELECT AVG(Salary)
         FROM Professor))
```

Figure 25.2: Alternative query for question E17— Whose salary is greater than the overall average salary?— showing the use of binary comparators in front of subqueries.

There is an important restriction on the use of aggregates: SQL does not allow what is conventionally called *double aggregation.* That means that it is syntactically illegal to nest aggregate functions— as in, AVG(SUM(...)), for example.

Consequently, it is impossible to pose, *in a single SQL statement,* a question such as: *What is the average of the departmental salary totals?* Rather, it can be posed in two steps: first,

retrieving salary sums into some table, and then computing the average over it. This, however, requires insert capabilities, which we will cover later.

In queries with subqueries, this prohibition against double aggregation also disallows the use of aggregate functions and/or GROUP BY or HAVING clauses in any two query blocks where one is included in the scope of the other. For example, it is permitted to use them in two subqueries if the subqueries are "next" to each other, but not if one is "inside" the other.

For those familiar with the concept of a *query tree*, a more precise way of saying this is: Given a query tree, it is prohibited to use aggregate functions and/or the GROUP BY or HAVING clauses in any two query blocks where one is a descendant of the other.

The unavailability of double aggregation also sometimes forces us to use non-aggregated formulations to pose questions that have better aggregation-based solutions. As an example, try posing in a single query with subqueries the question *What are the grade point averages of the youngest students?*

26 The Overall Order of Evaluation

One of the principal SQL programming skills is the ability to look at a complex query involving many different features, and to determine exactly in what order these features will be executed. As an example, we will consider the query of Figure 26.1, which involves most of the SQL features presented thus far.

```
SELECT DISTINCT Sname,
        "Grade Point Average" =
        round(SUM(Grade*Credits)/SUM(Credits),2)
FROM Student S, Take T, Course C
WHERE (S.Sno = T.Sno)
   AND (T.Cno = C.Cno)
   AND (S.Sno NOT IN
        (SELECT Sno
         FROM Take
         WHERE (Cno = "CS112")))
GROUP BY S.Sno, Sname
HAVING (SUM(Credits) > 12)
ORDER BY 2 DESC, Sname
```

Figure 26.1: A query involving most of the features presented so far.

This query corresponds to the following question:

What is the grade point average (GPA) of each student, who does not take CS112, but who takes more than 12 credits in total? Show the answer as student name/GPA pairs, without duplicates, and with GPA rounded to 2 decimal digits, sorted first in decreasing order by GPA, and then in increasing order by student name.

The overall order of events in evaluating this query is as follows:

1. Formulate and evaluate the underlying query:

```
SELECT *
FROM Student S, Take T, Course C
WHERE (S.Sno = T.Sno)
   AND (T.Cno = C.Cno)
   AND (S.Sno NOT IN
         (SELECT Sno
         FROM Take
         WHERE (Cno = "CS112")))
```

Since this is a Type 2 query, its subquery is executed first, followed by the main query, with the clauses considered in the following order:

1.1. the inner FROM clause:

```
FROM Take;
```

1.2. the inner WHERE clause:

```
WHERE (Cno="CS112");
```

1.3. the inner SELECT clause: SELECT Sno;
1.4. the outer FROM clause:

```
FROM Student S, Take T, Course C;
```

1.5. the outer WHERE clause:

```
WHERE  (S.Sno = T.Sno)
    AND (T.Cno = C.Cno)
    AND (S.Sno NOT IN
            (<answer to the subquery> ))
```

1.6. the outer SELECT clause: SELECT *.

2. Group the rows from the result of Step 1 by S.Sno and Sname, according to the GROUP BY clause:

```
GROUP BY S.Sno, Sname.
```

3. Evaluate the HAVING condition (SUM(Credits) > 12) for each group:

3.1. compute SUM(Credits) for every group;

3.2. eliminate those groups for which the HAVING clause condition fails.

4. Prepare the preliminary result according to the SELECT clause:

4.1. compute the multiplication (Grade*Credits) on *individual* Grade and Credits values for each row in each surviving group;

4.2. compute SUM(Grade*Credits) over the results of the multiplication from Step 4.1, and SUM(Credits) for each surviving group;

4.3. compute division in SUM(...)/SUM(...) and round the results to 2 decimal digits;

4.4. create a preliminary result with one row consisting of Sname and the just computed GPA expression for each surviving group.

5. Sort the preliminary result, first by the GPA expression descending, and then by Sname.

6. Eliminate duplicate rows, if any, from the preliminary result.
7. Label the second column of the result as "Grade Point Average" and stop.

This sequence of events was created by following the evaluation mechanisms presented earlier. As such, this is a "formal" sequence, created for the sole purpose of tracing. Any real SQL evaluation would certainly be more optimized— e.g., there is really no need to compute SUM(Credits) in Step 4.2 since it has already been computed in Step 3.1. In fact, the real order of evaluation may generally be quite different— we only guarantee that the results will be the same.

27 NULL Values in SQL

NULL is a special symbol that can be used in place of regular values in tables to mean that the value in question is currently unknown. (Other meanings of NULLs also exist, but they are much less common.) The SQL rules for handling NULLs are as follows.

NULLs can appear in any column that has been declared with the NULL qualifier in the CREATE TABLE statement— e.g., columns Sname and Age in table Student created in Figure 3.1.

All scalar arithmetic expressions involving at least one NULL evaluate to NULL. For example, given a Professor row with the NULL Age value, the expression (Age+1) evaluates to NULL. This is reasonable, because if an Age value is unknown, it follows that the result of (Age+1) is also unknown.

Most scalar expressions for other data types follow this rule as well, and return a NULL if at least one of their arguments is NULL. (There are some exceptions to this, particularly when dealing with strings; consult the appropriate SQL manuals for details.)

All binary comparisons involving the operators =, !=, <, <=, >, and >=, and at least one NULL evaluate to a new logical value *Maybe*. For example, given a Professor row with the NULL Age value, the condition (Age > 50) evaluates to Maybe. This is again reasonable, because we simply do not know whether his age is *definitely greater* or *definitely not greater* than 50.

Since traditional Boolean (two-valued) logic is not sufficient to support this new logical value Maybe, SQL supports what is known as the *three-valued logic*, defined by the truth tables shown in Figure 27.1. (In these tables, symbols F, M and T stand for False, Maybe and True, respectively.) We note that for the False and True combinations these tables are consistent with their traditional Boolean counterparts.

NOT	
F	T
M	M
T	F

AND	F	M	T
F	F	F	F
M	F	M	M
T	F	M	T

OR	F	M	T
F	F	M	T
M	M	M	T
T	T	T	T

Figure 27.1: Truth tables for the three-valued logic.

Since none of the standard binary comparators can be used to identify and test for NULLs, to recognize them SQL provides two special operators: IS NULL and IS NOT NULL, as in (Age IS NULL) and (Age IS NOT NULL).

The condition (Age IS NULL) returns True if the Age value is in fact NULL, and returns False if it is some regular— i.e., non-NULL— value. The condition (Age IS NOT NULL) acts in the opposite way. (In fact, the expression (Age IS NOT NULL) can be thought of simply as a shorthand for NOT (Age IS NULL).) Naturally, the left operand of the IS NULL and IS NOT NULL operators need not be a simple column reference but can also be an expression.

The behavior of the IN and NOT IN operators in the presence of NULLs is somewhat complex.

Given a regular value as its left operand, and a list of values containing at least one NULL on its right (in the answer to the subquery, that is), the IN operator returns True if that value is present in the list, and returns Maybe if it is not. (One of the NULLs in the list may actually stand for that regular value, and we do not definitively know whether this is or is not so.)

The NOT IN operator acts in the opposite way. Given a regular value on its left, and a list of values containing at least one NULL on its right, the NOT IN operator returns False if that value is present in the list, and returns Maybe if it is not.

Given a NULL as a left operand, and IN operator returns Maybe if the answer to the subquery is non-empty, and returns False if it is empty. Given a NULL as a left operand, the NOT IN operator again returns Maybe if the answer to the subquery is non-empty, but returns True if it is empty.

The behavior of the WHERE clause is refined to retrieve rows into the answer only when the WHERE condition evaluates to True; thus, within this clause, Maybe is treated in the same manner as False.

Similarly, the HAVING clause also treats Maybe in the same manner as False, and suppresses groups for which the HAVING condition evaluates to Maybe or to False.

The behavior of aggregate functions in the presence of NULLs is defined as follows. As long as at least one regular value exists in a column, the aggregate functions MIN(), MAX(), AVG() and SUM(), will operate only on the regular values, in effect ignoring the NULLs. Given a column consisting entirely of NULLs or given an *empty column*, all of

these functions return NULL. (The latter case happens when an aggregate is evaluated over the empty result returned by its underlying query.)

The aggregate function COUNT(DISTINCT <column>) also counts only the regular values. Thus, as long as at least one regular value exists in a column, this function will return the number of distinct regular values. Given the column of all NULLs or given an empty column, it will return 0 (zero).

However, the aggregate function COUNT(*) simply returns the number of rows in a group, regardless of whether or not some of the values are NULL.

In the GROUP BY clause, all NULLs are treated as equal. Thus,

```
GROUP BY Age
```

would group all rows with the NULL Age value into the same group.

For the purposes of duplicate suppression (in SELECT DISTINCT) all NULLs are also treated as equal.

In the ORDER BY clause, all NULLs are treated as smaller or larger than any other value, depending on the implementation (check the appropriate manual).

We note that there are some conceptual inconsistencies in the SQL's treatment of NULLs. First, the equality

```
SUM(Age)/COUNT(*) = AVG(Age)
```

no longer holds true, when evaluated for column Age comprised of some regular values and some NULLs. For example, given a column of Age values (50, 52, NULL), the expression SUM(Age)/COUNT(*) returns 102/3=34, while AVG(Age) returns 51.

Second, even though all NULLs may potentially stand for different values, the GROUP BY clause and the duplicate suppression mechanism in SELECT DISTINCT treat all NULLs as the same.

Finally, the fact that all binary comparisons involving NULLs evaluate to Maybe may sometimes result in counter-intuitive query answers. To illustrate this, consider the query of Figure 27.2.

```
SELECT *
FROM Professor
WHERE (Age = Age)
```

Figure 27.2. A query that will retrieve all professors— or will it?

When the condition of this query is evaluated for a Professor row with the NULL Age value, it will return Maybe, thus suppressing this row from the answer. While consistent with the SQL treatment of comparisons involving NULLs, this is arguably counter-intuitive. After all, why should it matter that the Age value is NULL? Even if we do not know what the actual Age value is for some professor, surely, it is equal to itself! (This happens, by the way, because SQL does not attempt to *syntactically simplify* conditions.)

Similarly, the conditions (Age != Age), (Age < Age), etc., also all evaluate to Maybe, instead of to False, as our intuition might suggest.

In concluding this section, we note that we have not addressed here any semantic issues involving NULLs—e.g., When is their use appropriate?, How do they affect database designs?— and have simply concentrated on their treatment in SQL. We do mention, however, that it is a commonly held view that NULLs should not be allowed to appear in primary key columns; hence, the use of the NOT NULL qualifier for column Sno in Figure 3.1.

28 Introduction to Type 3 SQL Queries

While there are very few questions that really require Type 3 queries— all of them too convoluted to be included in this essay— Type 3 queries form an essential part of SQL and provide an important alternative to the Type 2 formulations.

Consider the query of Figure 28.1.

```
SELECT Sno, Sname
FROM Student
WHERE ("CS112" IN
        (SELECT Cno
         FROM Take
         WHERE (Sno = Student.Sno)))
```

Figure 28.1: A sample Type 3 query.

This query uses the column reference Student.Sno in the subquery. This column reference cannot be bound locally— its prefix "Student." does not match table name Take— and, according to the SQL scope rules, it is instead bound to the outer table Student. This *non-local reference* (or, *correlation)* is precisely what makes this query Type 3 rather than Type 2. (Recall that in Type 2 queries all bindings must be local.)

We note that any attempt to evaluate this query "inside-out" (as was done with Type 2 queries) is now meaningless— Student.Sno is *undefined* as we go through the local table Take. Instead, this query must be evaluated "outside-in," as follows.

We first pick some row from the outer table Student. Once picked, this row "gives" Student.Sno its value, which "completes" the condition of the subquery, and, in effect, reduces the subquery to a simple Type 1 query.

We then evaluate the subquery for that value of Student.Sno, evaluate the IN operator in the main condition with respect to the subquery's result, and, if True, retrieve the Sno and Sname values from our picked Student row into the main result.

We then repeat this process for every other row from the outer Student table.

By following this trace, we can determine that this query asks our question E2— *What are student numbers and names of students who take CS112?*

Several things should be noted here. First, we do evaluate the subquery anew for every outer Student row.

Second, this is not terribly inefficient, since we are again describing a formal evaluation mechanism here, and thus the issue of efficiency does not even apply. (While the actual evaluation strategy taken by any real SQL system may be quite different, we do guarantee that it will always generate the same answer as our formal evaluation sequence.)

Third, the IN operator used here is the same old IN introduced earlier; we only note the use of a constant on its left, which is syntactically fine— we simply have not seen examples of such use earlier.

Fourth, we are not limited to just a single correlation. For example, the query of Figure 28.2 (which poses the question

Who teaches CS112?) uses two correlations: Professor.Fname and Professor.Lname, both of which get their values from the outer Professor rows.

```
SELECT Fname, Lname
FROM Professor
WHERE ("CS112" IN
        (SELECT Cno
         FROM Take
         WHERE (Fname = Professor.Fname)
            AND (Lname = Professor.Lname)))
```

Figure 28.2: A Type 3 query with two correlations, for the question Who teaches CS112?

29 Inter-Query Connectors Revisited: Existential Quantifiers in SQL

So far we have seen only two kinds of inter-query connectors: the list membership operator IN and, in limited circumstances, regular binary comparators (=, <, etc.). SQL also provides another inter-query operator, EXISTS, which syntactically acts as a *Boolean function*— i.e., it takes a subquery as an argument, and returns a Boolean as the result.

Semantically, the behavior of EXISTS is defined as follows: It returns True if the subquery retrieves at least one row in its answer, and returns False otherwise— i.e., when the answer to the subquery is empty.

An example of the use of EXISTS is shown in Figure 29.1.

```
SELECT Sno, Sname
FROM Student
WHERE EXISTS (SELECT *
                FROM Take
                WHERE (Sno = Student.Sno)
                  AND (Cno = "CS112"))
```

Figure 29.1: An example query with EXISTS.

We can now trace the query of Figure 29.1— after all, it is just a regular Type 3 query— and determine that it again poses our question E2— *What are student numbers and names of students who take CS112?* (We are intentionally staying with the same question here.)

Three things should be noted about EXISTS. First, *it is not the use of* EXISTS that makes the query Type 3— non-local references do. We make this point because, in practice, EXISTS and correlations are very often used together— indeed, they do compliment each other very well. However, they do not have to be used together. We may have Type 3 queries not involving EXISTS— e.g., the two queries from the previous section. Likewise, there are cases— albeit infrequent ones— where EXISTS is used in conjunction with Type 2 queries.

Second, by syntax, the SELECT clause of the subquery under EXISTS must contain just symbol * (asterisk). Putting anything else there will cause a syntax error.

Third, since EXISTS returns a Boolean result, it can be included into general logical expressions, and thus may have NOT, AND or OR connect it to other parts of the WHERE clause. (In fact, NOT EXISTS (...) formulations are very common in posing queries involving negation.)

Finally, for those familiar with the first order predicate calculus, EXISTS is an SQL implementation of the existential quantifier. Notably, SQL does not provide a similar FOR-ALL operator for the universal quantifier, which then has to be implemented via NOT-EXISTS-NOT formulations.

30 Some Old Questions Revisited— Again

In this section we look at an assortment of previously considered questions and develop their implementations using Type 3 queries. We begin with the queries just developed.

By changing IN to NOT IN in the query of Figure 28.1, we get our old question E5— *Who does not take CS112?* We also get this question by placing NOT directly in front of EXISTS in the query of Figure 29.1.

By changing IN to NOT IN in the query of Figure 28.2, we get the question *Who does not teach CS112?* (Try posing this question using NOT EXISTS yourself.)

We now consider the query of Figure 30.1, which shows an interesting implementation of question E9— *Who takes exactly 2 courses?*

```
SELECT X.Sno
FROM Take X, Take Y
WHERE  (X.Sno = Y.Sno)
    AND (X.Cno < Y.Cno)
    AND NOT EXISTS
            (SELECT *
            FROM Take
            WHERE (Sno = X.Sno)
                AND (Cno != X.Cno)
                AND (Cno != Y.Cno))
```

Figure 30.1: A Type 3 query for question E9 – Who takes exactly 2 courses?

This implementation is based on the fact that "exactly 2" means "at least 2 and not a third." In other words, a student takes exactly two courses if he takes at least two courses and there does not exist a third course (different from the first two) that he also takes.

Aliases X and Y in the outer FROM clause of this query are used not only to disambiguate the two outer copies of Take from each other, but also to bind references X.Sno, X.Cno and Y.Cno in the subquery to the outer copies of Take, thus allowing them to "escape" their local FROM clause.

The query of Figure 30.2 implements question E12— *Who are the youngest students?*

```
SELECT Sno
FROM Student S
WHERE NOT EXISTS
        (SELECT *
        FROM Student
        WHERE (Age < S.Age))
```

Figure 30.2: A Type 3 query for question E12 – Who are the youngest students?

This query is based on the observation that a student is youngest if there does not exist any student who is younger (than he is). We again note the use of aliasing in the outer query.

Type 3 facilities are also very useful in posing questions that compare base values with aggregates. For example, the query of Figure 30.3 implements question E18— *Whose salary is greater than the average salary within that professor's department?*

```
SELECT Fname, Lname
FROM Professor X
WHERE (Salary > (SELECT AVG(Salary)
                 FROM Professor
                 WHERE (Dept = X.Dept)))
```

Figure 30.3: A Type 3 query for question E18 — Whose salary is greater than the average salary within that professor's department?

This query also shows an example of using an aggregate (a *global aggrega*te, in this case) inside a subquery. The evaluation rules, however, remain exactly the same: this is a Type 3 query, so we execute it "outside-in"— we pick every professor in turn, supply his department (through X.Dept) to the subquery, execute the subquery thus computing the average salary in that department, and then compare his salary to that average.

We note that the treatment of aggregates and the use of the greater-than operator in front of the subquery are exactly as they were presented earlier in this essay. In fact, this is the whole point of this essay: Once you know the SQL building blocks and know how to connect them— you know the language.

Our final examples of Type 3 queries are two re-implementations of question Q13— *Who takes every course?* The first of them, shown in Figure 30.4, is interesting because it shows a three-level Type 3 query with two correlations: one between the outer query and the inner one, and the other between the middle query and the inner one.

```
SELECT Sno
FROM Student
WHERE NOT EXISTS
    (SELECT *
     FROM Course
     WHERE NOT EXISTS
         (SELECT *
          FROM Take
          WHERE (Sno = Student.Sno)
              AND (Cno = Course.Cno)))
```

Figure 30.4: A Type 3 query for question E13— Who takes every course?

To trace this query, we first pick some student from the outer Student table, and substitute his Sno (say 123456789) for Student.Sno in the inner WHERE clause. This would result in the middle query effectively becoming as shown in Figure 30.5. (Naturally, when done with this student, we will repeat the entire process for every other student in turn.)

```
SELECT *
FROM Course
WHERE NOT EXISTS
              (SELECT *
               FROM Take
               WHERE (Sno = "123456789")
                   AND (Cno = Course.Cno))
```

Figure 30.5: The middle query from Figure 30.4 after substitution for S.Sno.

Since this is itself a Type 3 query, we again evaluate it "outside-in." Specifically, we pick every course in turn, supply its number (through Course.Cno) to the inner subquery, execute the subquery, and then test if a row connecting that course to our student 123456789 was found in Take. The answer to this partial query then contains all columns for those courses which are not taken by our student 123456789.

Once the middle query terminates for student 123456789, the main WHERE clause then tests if it returned an empty answer. If yes, then there were no courses not taken by student 123456789, and he is retrieved into the final result. If no, then there were such courses— they are, in fact, the ones retrieved in the middle SELECT— and our student 123456789 is dropped from the final result.

Interestingly, the English reformulation of our question most closely corresponding to the query of Figure 30.4 is: *For which students, does there not exist a course that they do not take?*

The second Type 3 re-implementation of question Q13 is shown in Figure 30.6.

```
SELECT Sno
FROM Student
WHERE NOT EXISTS
        (SELECT *
         FROM Course
         WHERE (Cno NOT IN
                  (SELECT Cno
                   FROM Take
                   WHERE (Sno = Student.Sno))))
```

Figure 30.6: A Type 3 query with a Type 2 subquery for question E13 — Who takes every course?

The difference between this solution and the one of Figure 30.4 lies in the connection between the middle and inner subqueries. Here, the trace begins in the same way— we pick some student from the outer Student table, and substitute his Sno (say 123456789, again) for Student.Sno in the inner WHERE clause, resulting now in the middle query as shown in Figure 30.7.

```
SELECT *
FROM Course
WHERE (Cno NOT IN
        (SELECT Cno
         FROM Take
         WHERE (Sno = "123456789")))
```

Figure 30.7: The middle query from Figure 30.6 after substitution for S.Sno.

This query, however, is now of Type 2! Thus, it is evaluated "inside-out," and retrieves all information about courses that *are not among* those taken by student 123456789— or, rephrasing, those courses which *are not taken* by student 123456789. Since this gives the same intermediate result as the query of Figure 30.5, the rest of the trace proceeds as in our original query.

The English reformulation of our question that most closely corresponds to this query form is: *For which students, does there not exist a course which is not among the courses they take?*

31 Data Manipulation Facilities of SQL

The data manipulation language (DML) component of SQL provides facilities for inserting, deleting and updating rows in tables. There are two forms of insertions: one for inserting individual rows, and one for inserting all rows from query results. We begin with the former.

An example of a single row insertion is shown in Figure 31.1.

```
INSERT Student
VALUES ("123456789", "Robert Brown", 20)
```

Figure 31.1: Inserting individual rows.

The syntax is self-explanatory. We only emphasize that the argument to the VALUES keyword is a *single row* of values, presented in the *same order* as the order of columns in the corresponding table declaration.

The second form of insertion is much more interesting— as it turns out, an entire answer to an SQL query can also be inserted into a table. As an example, assume that we have created a table Seniors(Sno, Sname, GPA), and consider the query of Figure 31.2.

```
INSERT Seniors
SELECT S. Sno, Sname,
        round (SUM(Grade*Credits)/SUM(Credits),2)
FROM Student S, Take T, Course C
WHERE  (S.Sno = T.Sno)
    AND (T.Cno = C.Cno)
GROUP BY S.Sno, Sname
HAVING SUM(Credits) > 90
```

Figure 31.2: Inserting a query result.

Instead of just being displayed and discarded, the result of this query, which computes GPAs of all students who have taken more than 90 credits, is inserted into table Seniors.

Three things need to be noted in connection with the INSERT command. First, the target table must already exist.

Second, the INSERT command simply *appends* rows to a table and does not overwrite its previous content— if the target table already contained some previously inserted rows, they remain there. Thus, the INSERT ... SELECT construct does not act as an *assignment* operator, which *replaces* the previous content of its target.

Third, the query in the INSERT ... SELECT construct can be any SQL query. (There are some exceptions to this, but they involve SQL features not covered in this essay.) The only constraint here is that the *signatures*, as they are called— i.e., the number of columns and the order of their data types— of the target table and of the SELECT clause should correspond to each other. (In fact, the data types of the columns in the target table need not exactly match the data types of

the columns of the answer; they only have to "cover" them— e.g., the integer column in the SELECT clause can be inserted into a floating point column in the target table.)

Deletion of rows is done using the DELETE command, as shown in Figure 31.3.

```
DELETE
FROM Seniors
WHERE (GPA < 2.0)
```

Figure 31.3: Example of a DELETE command.

The syntax and the behavior of this statement is again self-explanatory— it will delete from table Seniors all rows with GPA less than 2.0. We only emphasize that the FROM clause here can involve only a single table, and that the WHERE clause can be any SQL WHERE clause, and in particular can involve subqueries.

We note that omitting the WHERE clause in the DELETE command has the same meaning as omitting it in a regular query— it is equivalent to the WHERE condition always being True. Thus, omitting the WHERE clause in this case would delete *all* rows from table Seniors. (So be careful— SQL provides no warnings!) We also note that table Seniors (or, more precisely, its structural definition) would still remain— albeit empty; to destroy it would require the use of the DROP TABLE command.

Updating rows is done using the UPDATE command, as shown in Figure 31.4, which in this case gives a 10% raise to all professors from the IS department.

```
UPDATE Professor
SET Salary = Salary*1.1
WHERE (Dept = "IS")
```

Figure 31.4: Example of an UPDATE command.

Again, several things need to be noted here. First, the WHERE clause can again be any SQL WHERE clause, and can involve subqueries. Omitting it is again equivalent to the WHERE condition always being True, causing all rows to be updated.

Second, the argument to the SET clause can involve multiple columns. For example, by modifying the SET clause in the above command to

```
SET Salary = Salary*1.1, Dept = "CIS"
```

we can change the name of the IS department simultaneously with giving IS professors the raise.

Third, it is allowed to mention the same column both in the WHERE and SET clauses; no confusion arises, because the WHERE clause is evaluated with respect to "old version" of rows, before any of them is actually updated.

The standard UPDATE command also suffers from one important limitation— while the condition on *which* rows to update may involve other tables in the database (through the use of subqueries in the WHERE clause), the *new value expressions* (i.e., the expressions to the right of the equal sign

in the SET clause) can only mention the columns from the table being updated. Thus, it is not possible to update a column in one table based on the values from some column(s) in some other table.

To compensate for this limitation, some SQL dialects have extended the UPDATE command by permitting a FROM clause (placed between the SET and WHERE clauses), and by allowing columns from all of the tables in that FROM clause to be used in the new value expressions. (Consult appropriate SQL manuals for details.)

32 Extensions to SQL

To appreciate the need for and the added value of SQL extensions, it is important to understand that, when originally conceived, SQL was viewed not as a full programming language, but only as a *data sublanguage.* It was meant to be used either *interactively,* allowing end-users to pose their queries one at a time, or to be used in a larger programming system, by *embedding* SQL queries into some conventional programming language.

Consequently, it was considered unnecessary to provide SQL with facilities that were not likely to be used by users in the interactive mode, or that were already available as conventional features in the host programming language. Thus, the "classical" SQL— which is what we have presented in this essay— lacks both the standard control structures and the complex data structuring mechanisms.

In the recent years however, there has been a concerted effort on the part of the vendors to extend SQL and to make it more like a complete programming language. This is a positive development as SQL popularity grows and as it becomes recognized that embedded SQL systems are not well suited for *client-server* architectures, as they often create massive communication overhead between the database server and its clients.

Some of the extensions that significantly increase the expressive power of SQL are:

- the *standard control structures* (e.g., WHILE and FOR loops, and IF-THEN-ELSE statements) and *compound statement constructors* (e.g., BEGIN and END), that operate on SQL query blocks;
- *scalar variables,* along with the ability to use them in the SELECT clause to get values from queries, and in the WHERE and HAVING clauses to supply values to queries;
- *procedures* (usually called *stored procedures*), that usually allow for both *in* and *out* parameters, and that allow for segments of SQL code to be grouped together and pre-compiled;
- *user-defined data types,* along with the ability to define related scalar operators and functions, which can then be used in the clauses of SQL queries.

These extensions are not yet standard; in fact, they are not even taken from any single system, but are representations of what is available in many different SQL dialects. However, they bring SQL closer to a full programming language, and allow many problems not previously solvable within SQL to now have elegant and efficient solutions. Further discussion of these extensions, as well as of the ongoing efforts to standardize the language, is beyond the scope of this essay.

33 Last Remarks

We now come to the end of this essay. If you, the reader, have read it in order, and actually traced every query presented, then you should now be comfortable with SQL.

Yes, it is true that we have skipped some of the features of SQL. But SQL is an evolving language— so any attempt to provide a *complete* yet *lasting* coverage of it in any single presentation is doomed anyway. Rather, we wanted to present the fundamental features of SQL— the ones that define its *essense.*

By motivating these features with specific standard question types, by explaining inter-relationships among them, and by showing various standard solutions for these questions, we wanted to give you a solid foundation— a "culture," if you will— for both the use and further study of this important, powerful and elegant language.

Given this foundation, you should now have no trouble understanding all of those features of SQL that we did not cover— both in terms of what they mean, and how to use them. (This understanding may even extend to those SQL features not yet invented.) You should also have no trouble adapting your understanding of the generic SQL presented in this essay to any real SQL dialect supported by an actual database management system.

In presenting SQL, we have concentrated exclusively on the issues of query *correctness* rather than their *efficiency,* the latter being reserved for a separate essay. This, in turn, allowed us to reduce the enormous complexity of the actual

evaluation strategies used by real SQL systems to four simple formal evaluation mechanisms, bringing clarity and conciseness to the language and its presentation.

We leave you with a list of assorted reminders, shown in Figure 33.1, for programming in SQL. The items in this list were chosen for two basic reasons: first, they are important, and second, people tend to forget them.

---

1. Do not rely on your intuition in composing or interpreting SQL queries; trace them according to the evaluation mechanisms:
   - remember the order of evaluation;
   - remember that formally Type 2 queries are evaluated inside-out;
   - remember that formally Type 3 queries are evaluated outside-in;
2. Remember that real negation requires two passes: To find out "who does not" first find out "who does" and then get rid of them.
3. Remember that NOT appearing in front of the subquery cannot be brought inside its WHERE clause; doing so changes the overall query meaning.
4. Remember that in queries involving aggregate functions and/or GROUP BY, only those columns explicitly listed in the GROUP BY clause may appear un-aggregated in the SELECT clause and in the HAVING clause.

Figure 33.1: Assorted reminders for programming in SQL.

Acknowledgments

I wish to express my thanks to my colleague, Professor Susan Dorchak, for her very substantial help in reviewing and editing this essay. I also wish to express my gratitude to my former student, Helen Zeldovich, for her invaluable assistance in improving the clarity of this presentation and for her tireless help in proofreading this essay.

Notes

Notes

Notes

Notes

Notes

DATABASE TECHNOLOGY
LANGUAGES

The Essence of SQL:

A Guide to Learning Most of SQL in the Least Amount of Time

by David Rozenshtein

This book is ideal for programmers, managers, and students who want to quickly acquire an in-depth understanding of SQL. It takes the reader from the fundamentals of SQL through to the most complex features of the language. It explains why the various constructs in SQL exist and how SQL can be used to solve real-world, business problems.

The Essence of SQL provides a practical, problem-centered approach to the study of this powerful database language. It concentrates on those aspects of SQL which have traditionally presented problems to programmers; it promotes a smooth transition to SQL from other programming languages. This book allows the reader to gain a clear understanding of the essence of SQL in a minimum amount of time. The material presented in the book applies to all SQL dialects.

- ✔ ***Programmers*** will find the many SQL code examples to be useful building blocks for designing effective relational database solutions.
- ✔ ***Managers*** will find a clear description of the power of SQL and guidance on its appropriate uses in relational database systems.
- ✔ ***Students*** will find the book to be an invaluable study tool and accelerated, rigorous SQL tutorial.

About the Author:

David Rozenshtein, Ph.D., is an Associate Professor of Computer Science at Long Island University and an independent consultant specializing in the design, implementation, and optimization of large-scale, SQL-based systems; he has lectured extensively on SQL and other database topics; he is the author of 2 books and many technical articles; he holds a Ph.D. in Computer Science from the State University of New York at Stony Brook.

ISBN 0-9649812-1-1

SQL FORUM PRESS
40087 Mission Boulevard, Suite 167