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.

Type 1 Queries *or* Type 2 Queries *or* Type 3 Queries
Aggregation Facilities
Enhancements
Extensions

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.