CS 252: Fundamentals of Relational Databases: SQL5

Dr. Alexandra I. Cristea
http://www.dcs.warwick.ac.uk/~acristea/

Careful study of these notes is best left until most of the lectures on CS252 have been given. Many of these notes relate SQL to the theory section of CS252. In some cases they refer to things you will encounter in the theory section after this lecture has been given.
Interim Summary

Topics covered:
• Indexes, Sequences,
• Relational Algebra and Advanced Joins
This lecture

• Query re-write
• NULLs and 3-valued logic
• Top N Analysis
• Security Privileges
The **CASE** operation returns one of a specified set of scalar values depending on some condition.

```
CASE
WHEN CD_Year.year >= 2000 THEN 'Noughties'
WHEN CD_Year.year >= 1990 THEN 'Nineties'
WHEN CD_Year.year >= 1980 THEN 'Eighties'
WHEN CD_Year.year >= 1970 THEN 'Eighties'
ELSE 'Before my time'
END
```

The CASE construct is found in most programming languages. You are possibly familiar with it already. Remember that the individual WHEN cases are evaluated in the order in which they are written and the result is the value of the expression given in THEN clause accompanying the first WHEN condition that results in *true*. If each of the WHEN conditions results in either unknown or false, then the result is the value of the expression given in ELSE clause.

With SQL's counterparts of well-known logical constructs, you need to consider the implications of SQL's treatment of its idiosyncratic third truth value. Note that `CASE WHEN a = b THEN c ELSE d` and `CASE WHEN NOT (a = b) THEN d ELSE c` are not equivalent in SQL! This is because `a = b` and `NOT (a = b)` both evaluate to *unknown* when either `a` or `b` is NULL.
Redundancy of GROUP BY

For every select expression that involves `GROUP BY` or `HAVING` there is an equivalent expression that does not.

```
SELECT student, SUM(mark) Total FROM CS_marks GROUP BY student;
```

can be re-written to be:

```
SELECT DISTINCT student,
    ( SELECT SUM(mark) FROM CS_marks CSM
        WHERE CS_marks.student = CSM.student ) Total
FROM CS_marks;
```

Redundancy in a computer language is pretty well unavoidable and in any case is to some extent to be desired. For example, we don't really need both of the comparison operators `<` and `>`, but we certainly wouldn't want to discard one of them. It is perhaps unfortunate, though, when redundancy arises from corrections and additions to the language that become needed simply because the language was badly designed to begin with. Here we can note that the original SQL of 1979 included `GROUP BY` but did not support the alternative expression shown on this slide, because in 1979 subqueries were allowed only in the `WHERE` and `HAVING` clauses and not, for example, the `SELECT` clause. The term orthogonality refers to that aspect of good language design whereby an expression can be used wherever the effect of it makes sense. The expression in question here is the subquery beginning `SELECT SUM(mark)`, which, although it looks like a table expression, here denotes a number (we say that the table is coerced to a number, coercion being a dangerous and deprecated practice that can nevertheless afford a certain amount of convenience). Of course any expression denoting a number should be allowed to appear as an element in the `SELECT` clause of a query.

Original SQL was subject to much criticism for its lack of orthogonality. Nowadays it is much more orthogonal, but the corrections have resulted in perhaps more redundancy than would normally be desired. Too much redundancy leads to too much choice. Why too much? Well, consider that the various choices might have different performance characteristics, and that the fastest solution in SQL implementation A is not always the fastest in implementation B. Making the "right" choice can sometimes be too demanding.

Recall that Tutorial D's GROUP operator — which is related to SQL's `GROUP BY` even though it is not the same operator — is also redundant, being defined in terms of EXTEND and projection.

You can think of Tutorial D's GROUP and SQL's `GROUP BY` as being in a sense opposites of each other. Using the example shown on the slide, assuming `CS_marks` has just the two columns, Student and Mark, GROUP BY Student would become `GROUP \{ \langle Mark \rangle \}` in Tutorial D.

Actually, Tutorial D does have a direct counterpart of `GROUP BY` in the `BY \{ \ldots \}` clause of its SUMMARIZE operator, but `BY \{ \ldots \}` is not supported by Rel at the time of writing (October 2007) and you have to use `SUMMARIZE PER` instead.
Example with HAVING

```sql
SELECT student, SUM(mark) AS Total FROM CS_marks GROUP BY student
HAVING SUM(mark) > 100;
```

can be re-written as

```sql
SELECT student, Total
FROM ( SELECT student, SUM(mark) AS Total
       FROM CS_marks
       GROUP BY student ) t
WHERE Total > 100
```

Actually it is obvious that HAVING is redundant, because it means exactly the same as WHERE! The difference between the two is a matter of (bad) syntax, not semantics. A WHERE clause operates solely on the result of a FROM clause, whereas HAVING operates solely on the result of a GROUP BY clause. (It is true that a HAVING clause can immediately follow a FROM clause or a WHERE clause, but when it does, GROUP BY ( ) — i.e., group by no columns at all — is implicit.) So the example shown on the slide can also be expressed this way:

```sql
SELECT student, Total
FROM ( SELECT student, SUM(mark) AS Total
       FROM CS_marks
       GROUP BY student ) t
WHERE Total > 100
```
or as:

```sql
SELECT DISTINCT student,
       ( SELECT SUM(mark) FROM CS_marks CSM WHERE CS_marks.student = CSM.student ) AS Total
FROM CS_marks WHERE
       ( SELECT SUM(mark) FROM CS_marks CSM WHERE CS_marks.student = CSM.student ) > 100;
```

It seems quite reasonable to ask, why bother with HAVING at all when it is so easy to achieve the same effect using WHERE and the rules governing the two are so unusual? As before, the reason lies in the history of SQL. Original SQL did not allow subqueries to appear in the FROM clause, a restriction that is still found in some implementations to this day. That restriction was a very bad mistake indeed. It resulted in SQL being relationally incomplete, even though its inventors thought they had shown it to be complete. Incompleteness means that certain problems that you should in theory be able to solve cannot actually be solved. Much worse, it also means that it can be very difficult to distinguish the solvable from the unsolvable — the user might waste a great deal of time looking for a solution that doesn't exist.
NULL

What does NULL mean?
SQL represents the fact that some piece of information is missing by the use of a special marker 'NULL'.

Mark of student Tim is NULL
This means:
• We know that Tim exists
• We know that Tim has a mark
• We don't know what the mark is

As the slide indicates, SQL allows the fact that a piece of information is missing to be denoted by the appearance of NULL in the place where a value denoting that piece of information would appear were it not missing. Note carefully that NULL is not a value, in the normal meaning of that term, even though it is often referred to as "the null value". It is not a value because it does not have the usual property of being equal to itself (except in certain circumstances—it is a very confusing construct).

Note also that the reason why Tim's mark is missing is just one of many different possible reasons. Perhaps Tim hasn't even taken the exam yet. Critics of SQL's invention of NULL, and of its behaviour when operated on by operators that are normally defined only for values, are fond of listing as many different reasons as they can think of, for why a piece of information might be "missing". The record at one time in the 1980s was 19 but has probably been beaten since then.

Although SQL "allows" NULL to be used for the purpose just mentioned, it actually "allows" it to be used for any purpose you might wish to use it for (how could it stop you?). Moreover, SQL itself actually does use NULL for purposes other than indicating that a piece of information is missing. For example, SELECT SUM(x) AS total FROM T yields NULL for the column total in the case where the table T is empty. The sum of no values isn't a piece of missing information—it is zero!
NULL and scalar expressions

If x or y (or both) is NULL then

\[ x + y \]
\[ x - y \]
\[ x \times y \]
\[ x / y \]

all evaluate to NULL

In some SQL implementations \( x/y \) can result in NULL when \( x \) and \( y \) are both numbers, but \( y \) is zero! So another meaning for NULL, used sometimes by SQL itself, is "undefined".
NULL and aggregate functions

- Aggregate functions ignore NULL's apart from COUNT(*).
- So SUM(column) is not the same as adding up all the values in the column with '+'.
- Aggregate functions where the input column contains no values return NULL.
- COUNT(*) returns 0 when counting no rows.

The treatment of aggregate functions in the absence of a GROUP BY clause (or equivalently, when GROUP BY () is specified explicitly) is somewhat inconsistent when the input table is empty. Consider SELECT c1, SUM(x) FROM t GROUP BY c1. If t is empty, the result is also empty. Contrariwise, the result of SELECT SUM(x) FROM t (equivalently SELECT c1, SUM(x) FROM t GROUP BY ()) always yields a one-row table, even when t is empty. That explains how COUNT(*) might sometimes evaluate to zero.
NULL and three-valued logic

SQL conditions can evaluate to **true**, **false** or **unknown**. This is a three-valued logic and our logic operators AND/OR/NOT need to be defined over all 3 values :

<table>
<thead>
<tr>
<th>AND</th>
<th>t</th>
<th>u</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>u</td>
<td>f</td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td>u</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>

t = true
u = unknown
f = false
The operators defined on this slide have names AND, OR, and NOT, and indeed, they have the same meaning as the familiar operators of those names when they operate on *true* and *false*. But the operators of SQL's logic are not the classical operators of those names and many of the theorems that hold in classical logic do not hold in SQL. For example, in classical logic $p \text{ OR NOT } (p)$ always evaluates to *true*, regardless of the truth value of $p$. In SQL $p \text{ OR NOT } (p)$ can evaluate to either *true* or *unknown*. A similar comment applies to $p \text{ AND NOT } (p)$, which in classical logic always evaluates to *false*. Because SQL uses the same operator names as classical logic and yet the theorems of classical logic do not hold in SQL, many people, including DBMS developers and, notably, developers of optimizers, have fallen into some of the traps that arise from NULL and 3VL. When an optimizer transforms an expression into one that is not equivalent to it, then the DBMS can give incorrect answers to queries, of course.
Default Column Values

When defining a table using CREATE TABLE the default value of a column is NULL. This can be changed in the CREATE TABLE command:

```sql
CREATE TABLE CS_marks
(
    student VARCHAR(20),
    course CHAR(3),
    mark INTEGER DEFAULT 50
);
```

```sql
INSERT INTO CS_marks (student, course) VALUES ('Paul Smith', 'DBS');
```

Inserts the row ('Paul Smith', 'DBS', 50) into the CS_marks table.

Actually, NULL is the "default default value". (Although NULL is not a value, it is quite difficult to talk about it without at times pretending that it is!) In other words, if no explicit DEFAULT clause is included in a column definition, then DEFAULT NULL is assumed. But you might prefer to write DEFAULT NULL explicitly, and that might be a good idea, reassuring the reader of the CREATE TABLE statement (perhaps the person marking your exam!) that you had thought about the matter and come to a deliberate decision on it.
TOP N Analysis

We can sort the results of a SELECT statement with
ORDER BY and ORDER BY ... DESC

Once we have defined an ORDER we may only want to return the top 10 records.

In SQL Server you can write
SELECT TOP 10 FROM CS_marks ORDER BY mark

In MySQL you can write
SELECT FROM CS_marks ORDER BY mark LIMIT 10

The syntax shown here is not in the international standard and is somewhat contentious. This is because there might be a tie for the \( n \)-th place, and yet the SQL DBMS will never yield more than \( n \) rows in the result of such a query. When there is a tie, the DBMS selects one of the participants arbitrarily. Thus, expressions using TOP \( n \) are in general non-deterministic (or indeterminate)—as is ORDER BY itself for that matter.
In Oracle things are more complicated. When Oracle evaluates a select query it appends a column to the result called 'ROWNUNM' before any ORDER BY, GROUP BY or DISTINCT clauses are run. So the query:

```
SELECT Student, Mark, ROWNUM FROM CS_marks;
```

Gives:

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>MARK</th>
<th>ROWNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul Smith</td>
<td>43</td>
<td>1</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>57</td>
<td>2</td>
</tr>
<tr>
<td>Helen Treacy</td>
<td>72</td>
<td>3</td>
</tr>
<tr>
<td>Paul Smith</td>
<td>65</td>
<td>4</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>42</td>
<td>5</td>
</tr>
</tbody>
</table>
while
SELECT Student, Mark, ROWNUM FROM CS_marks
ORDER BY Mark;

Gives:

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>MARK</th>
<th>ROWNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rachel Sewell</td>
<td>42</td>
<td>5</td>
</tr>
<tr>
<td>Paul Smith</td>
<td>43</td>
<td>1</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>57</td>
<td>2</td>
</tr>
<tr>
<td>Paul Smith</td>
<td>65</td>
<td>4</td>
</tr>
<tr>
<td>Helen Treacy</td>
<td>72</td>
<td>3</td>
</tr>
</tbody>
</table>

Which isn't very helpful, but if we force Oracle to append the ROWNUM after it has done the order then we can use it to limit the rows we get.
SELECT Student, Mark, ROWNUM FROM
(SELECT Student, Mark, ROWNUM FROM CS_marks ORDER BY Mark);

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>MARK</th>
<th>ROWNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rachel Sewell</td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>Paul Smith</td>
<td>43</td>
<td>2</td>
</tr>
<tr>
<td>Rachel Sewell</td>
<td>57</td>
<td>3</td>
</tr>
<tr>
<td>Paul Smith</td>
<td>65</td>
<td>4</td>
</tr>
<tr>
<td>Helen Treacy</td>
<td>72</td>
<td>5</td>
</tr>
</tbody>
</table>

What about?:
SELECT Student, Mark, ROWNUM FROM
(SELECT Student, Mark FROM CS_marks ORDER BY Mark);

Just by looking at this query, what do you think the result will be? Try it out on mimosa and check if you are right!
SELECT Student, Mark, ROWNUM FROM
(SELECT Student, Mark, ROWNUM FROM CS_marks
ORDER BY Mark)
WHERE ROWNUM < 3;

Gives us the lowest two marks:

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>MARK</th>
<th>ROWNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rachel Sewell</td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>Paul Smith</td>
<td>43</td>
<td>2</td>
</tr>
</tbody>
</table>
SQL Syntax for Security Rules

```
GRANT [privilege-commalist | ALL PRIVILEGES]
ON object-name
TO [authorisation_id_list | PUBLIC]
[WITH GRANT OPTION]
```

Each privilege is one of the following:

- SELECT
- DELETE
- INSERT [ (attribute-commalist)]
- UPDATE [ (attribute-commalist) ]
- REFERENCES [ (attribute-commalist) ]

The REFERENCES allows privileges to be granted on named table(s) in integrity constraints of CREATE TABLE.

The GRANT OPTION allows the named users to pass the privileges on to other users.

In the 1990s and 2000s many additional kinds of privileges arrived in standard SQL, in association with support for user-defined operators and user-defined types, for example. They are all beyond the scope of CS252 and the ones we teach here are as in the SQL of 1979.
Exercise:

Base relation STATS looks like this:
STATS (USERID, SEX, DEPENDENTS, JOB, SALARY, TAX, AUDITS)
PRIMARY KEY (USERID)

Write rules for the following:
(a) User John SELECT privileges over the entire table.
(b) User Fred UPDATE privileges over the TAX column only.
(c) How would you grant user Pope full privileges over rows for job type 'Priest' only?

Check them out:
• GRANT SELECT ON STATS TO John
• GRANT UPDATE(TAX) ON STATS TO Fred
• GRANT ALL(JOB='Priest') ON STATS TO Pope
  WITH GRANT OPTION
Grant and Revoke

If a user A grants privileges to user B, then they can also revoke them e.g.

```
REVOKE ALL PRIVILEGES ON STATS FROM John;
```

SQL REVOKE syntax
```
REVOKE [GRANT OPTION FOR]
[privilege_list | ALL PRIVILEGES]
ON object_name
FROM [authorisation_list|PUBLIC] [RESTRICT|CASCADE]
```

If RESTRICT option is given then the command is not executed if any dependent rules exist i.e. those created by other users through the WITH GRANT OPTION. CASCADE will force a REVOKE on any dependent rules.
Listing your security privileges

Under Oracle you can list your security privileges by using

```
SELECT * FROM SESSION_PRIVS;
```

<table>
<thead>
<tr>
<th>PRIVILEGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE SESSION</td>
</tr>
<tr>
<td>ALTER SESSION</td>
</tr>
<tr>
<td>CREATE TABLE</td>
</tr>
<tr>
<td>CREATE CLUSTER</td>
</tr>
<tr>
<td>CREATE SYNONYM</td>
</tr>
<tr>
<td>CREATE VIEW</td>
</tr>
<tr>
<td>CREATE SEQUENCE</td>
</tr>
<tr>
<td>CREATE DATABASE LINK</td>
</tr>
</tbody>
</table>

8 rows selected.

The table name used in this example refers to an example of what is commonly called a "catalog table" (or perhaps "catalogue table" on this side of the Atlantic). The international standard includes a standard catalog but numerous SQL implementations, including Oracle, already included proprietary catalogs of their own by the time the standard catalog appeared. SESSION_PRIVS is not part of the standard catalog, and nor, for that matter, is the information it conveys, for the standard does not include any statements for controlling who is allowed to CREATE things.
Interim Summary

- Query re-write
- NULLs and 3-valued logic
- Top N Analysis
- Security Privileges