Information Systems

WS 2005, JKU Linz
Course 6: SQL

Gábor Bodnár

URL: `http://www.risc.uni-linz.ac.at/education/courses/ws2005/is/`

Overview

- Data definition.
- Simple Queries.
- Views and Joins.

About SQL

SQL stands for Structured Query Language.

It is a standard, which was originally developed by IBM Research.

The current version is SQL 2003.

SQL specifies:

- a data definition language (DDL),
- a data manipulation language (DML),
- embedded SQL (to make relational databases accessible in other programming languages, like C, Pascal, PL/I).

About SQL (continued)

SQL also supports

- basic integrity enforcing,
- user authorization,
- transaction management.

In SQL terminology a relation is a *table*, and an attribute is a *column* and a tuple is a *row*.

Keywords of the language are capitalized structure is marked by EBNF notation.

Data Definition

A part of SQL is a data definition language (DDL), that allows us to

- define/modify/delete tables with prescribed domains for attributes,
- define or drop indices,
- specify integrity constraints,
- define authorization information.

Standard Types

Domain types for columns (i.e. attributes):

char(n) a character string of fixed length n,

int an integer (length can be implementation/hardware dependent),

numeric(i, d) a numerical value with i digits in the integer part (and a sign) and d digits in the decimal part,

real a single precision floating point number,

date storing the years in four digits and the months and the days in two,

time in hh:mm:ss format.

Standard Types (continued)

Coercion between compatible types, and the usual operations (e.g. arithmetic for numerical types, or string concatenation) are supported.

Many DBMSes also support the BLOB type (Binary Large OBject).

Simple domain definitions can be made, for example:

CREATE DOMAIN name_type AS CHAR(20)

Creating Relation Schemes

Creating a table with a given relation schema:

```
CREATE TABLE table_name (
        col_name col_type  [NOT NULL]
    [,  col_name col_type  [NOT NULL]]*
    [,  integrity_constraint])
```

Example

```
CREATE TABLE Student (
        sid       numeric(7,0)   NOT NULL
    ,   fname     name_type
    ,   lname     char(20)       NOT NULL
    ,   PRIMARY KEY (sid))
```

Example

Integrity constraints can be expressed with the CHECK clause.

```
CREATE TABLE Attendance (
        sid        numeric(7,0)    NOT NULL
    ,   cid        numeric(6,0)    NOT NULL
    ,   PRIMARY KEY (sid,cid)
    ,   CHECK   (sid IN (SELECT Student.sid FROM Student)) AND
                (cid IN (SELECT Course.cid FROM Course)))
```

Here we checked integrity constraints of foreign keys with a very explicit construction.

Example (continued)

In PostgreSQL this frequent problem can be treated with a language construction that specifies explicitly that something is a foreign key for another table.

```
CREATE TABLE Attendance (
      sid       char(7)          REFERENCES Student (sid)
    , cid       char(6)          REFERENCES Course (cid)
    , PRIMARY KEY (sid,cid)
```

The referenced columns must form a candidate key in the referenced table, and if none is given, the primary key is assumed by default.

Modifying and Deleting Relation Schemes

A table (relation scheme) can be modified with the ALTER TABLE clause.

```
ALTER TABLE table_name [ADD col_name col_type] |
    [DROP col_name]
```

A table can be deleted with the DROP TABLE clause.

```
DROP TABLE table_name
```

Creating and Dropping Indices

An index can be created with the CREATE INDEX statement.

CREATE INDEX index_name ON table_name (col_name [,col_name]*)

And it can be deleted by the DROP INDEX statement.

DROP INDEX index_name

Example

This defines an index with name “stid” on the `Student` table by the attribute `sid`.

CREATE INDEX stid ON Student (sid)

Simple Queries

In its simplest form the SELECT statement retrieves all the rows in the projection of the Cartesian product of the named tables to the specified columns.

```
SELECT [DISTINCT | ALL] [table.]column [, [table.]column ]*
        [INTO new_table_name]
        FROM table [, table ]*
```

The optional parameter DISTINCT results elimination of multiple rows (getting 1NF), while ALL forces to leave all rows in the result.

All the columns of a table can be denoted by an asterisk.

The INTO clause can specify the target table in which the results should be saved.

The WHERE clause

The WHERE clause can appear in context of a SELECT, DELETE, UPDATE clause.

WHERE [NOT] [table.]column operator [value | [table.]column]
        [AND | OR [NOT] [table.]column operator
        [value | [table.]column]]*

The “operator” can be any operator of the language which is applicable to the types of the columns appearing on its sides.

“Value” can be any literal value belonging to a type of the language that makes sense in context of the “operator”.

The ORDER clause

The ORDER clause appears always in context of a SELECT clause.

```
ORDER BY [table.]column [ASC | DESC]
        [, [table.]column [ASC | DESC]]*
```

Example

```
SELECT s.sid, s.fname, s.lname
        FROM Student AS s, Attendance AS a
        WHERE s.sid = a.sid AND a.cid = 327456
        ORDER BY s.lname ASC, s.fname ASC
```

This gets the ID and names of students attending the course with ID "327456" in a list ordered by last names and first names.

Set Theoretic Operations

- IN stands for set membership test.
- UNION and INTERSECT.
- EXCEPT stands for the set theoretic difference.
- EXISTS tests if the result of a query is nonempty.

The operators UNION, INTERSECT and EXCEPT eliminate duplicate rows by default.

Nested Subqueries (Example)

Selecting names of students who attend at least one course.

```
SELECT fname, lname
    FROM Student
    WHERE sid IN (   SELECT Student.sid
                     FROM Student, Attendance
                     WHERE Student.sid = Attendance.sid)


SELECT fname, lname
    FROM Student
    WHERE EXISTS ( SELECT *
                   FROM Attendance
                   WHERE Student.sid = Attendance.sid)
```

Grouping and Aggregate Functions

Aggregate functions like MIN, MAX, AVG, SUM, COUNT are supported by SQL.

Null values are eliminated from the columns before the function is applied.

The GROUP BY clause defines the groups for the aggregate functions and the HAVING clause eliminates aggregated values from the result which do not fulfill its condition.

If a WHERE clause is also present, its conditions are applied before the grouping is done.

Example

Selecting those students who have better averages than 2.

```
SELECT sid, AVG(grade)
        FROM Result
        GROUP BY sid
        HAVING AVG(grade) <= 2
```

Counting for each course the number of people passed the exam.

```
SELECT cid, COUNT(DISTINCT sid)
        FROM Result
        WHERE grade < 5
        GROUP BY cid
```

The DISTINCT keyword eliminates grade improvements (e.g. $4 \rightarrow 2$)

Database Modification

The structure of the statements are the same as for queries.

```
DELETE FROM table_name
        WHERE condition


INSERT INTO table_name [(col_name [, col_name]*)]
        [VALUES (value [,value]*) | select_cause]


UPDATE table_name
        SET col_name = expression [, col_name = expression]*
        WHERE condition
```

Example

```
INSERT INTO Student
        VALUES (0245768, 'Alexander', 'Wurz')
```

After the table definition

```
CREATE TABLE Namelist (
        fname     char(20)
    ,   lname     char(20))
```

we can fill it with

```
INSERT INTO Namelist
        SELECT fname, lname FROM Student
```

Example

Adding a student whose first name is not yet known.

```
INSERT INTO Namelist (lname)
        VALUES (’Hackl’)
```

Then later . . .

```
UPDATE Namelist
        SET fname = ’Hans’
        WHERE fname IS NULL AND lname = ’Hackl’
```

Deleting all students whose last names do not start with R or S.

```
DELETE FROM Namelist
        WHERE NOT (lname LIKE ’R%’ OR lname LIKE ’S%’)
```

Views

Views can be created with

CREATE VIEW view_name AS query

As discussed before updating the database via views can lead to problems, thus most DBMSes allow such actions only if the view was created using a single table.

A view can be canceled by

DROP VIEW view_name

Joins

One can construct joins in several ways in SQL. The general syntax of the join clause is

```
table_name {{ LEFT | RIGHT } [OUTER] | NATURAL |
        [FULL] OUTER | [INNER] } JOIN table_name
        { ON condition | USING(col [,col]*) }
```

The condition of the ON subclause can specify the columns on which the join is to be taken (e.g. when the column names are different).

With the USING subclause one can specify the columns on which the join is to be taken.

Example

Let us create the table ($\texttt{Student} \bowtie \texttt{Attendance}) \bowtie \texttt{Course}$.

```
CREATE TABLE StudentCourse (
      sid       numeric(7,0)
   ,  fname     char(20)
   ,  lname     char(20)
   ,  cid       numeric(6,0)
   ,  title     char(30))
```

Example

And fill it up with the data

```
INSERT INTO StudentCourse
    SELECT *
    FROM (Student INNER JOIN Attendance ON
            Student.sid = Attendance.sid)
            INNER JOIN Course USING(cid)
```

This view on the `StudentCourse` table eliminates IDs.

```
CREATE VIEW StCr AS
    SELECT fname, lname, title
    FROM StudentCourse
```

Embedded SQL

Embedded SQL is defined to allow access to databases from general purpose programming languages (Perl, C, Ada, etc.) which are called *host languages*.

The SQL statements in the host language are enclosed in an EXEC SQL, END-EXEC (or simply semicolon) pair.

Inside an SQL statement variable names of the host language can be referred by attaching a colon as a prefix.

The program in the host language that uses embedded SQL must also contain an SQLSTATE variable, in which the status code is returned after every SQL statement.

Example

Assume the “id” host variable contains a student ID whose name we want to fetch into “fn” and “ln”.

```
EXEC SQL
    SELECT fname, lname
        INTO :fn, :ln
        FROM Student
        WHERE sid = :id ;
```

Cursors

In embedded SQL the program in the host language can access the rows of the result one-by-one, and the mechanism to support this is called *cursor*. For each query one should define a cursor as

```
EXEC SQL DECLARE cursor_name CURSOR FOR query ;
```

and then retrieve the rows of the result in some kind of loop until SQLSTATE reports the end of data event and then close the cursor.

```
EXEC SQL OPEN cursor_name ;
do {
    EXEC SQL FETCH cursor_name INTO host_var [, host_var]* ;
}while(SQLSTATE != EndOfDataCode);
EXEC SQL CLOSE cursor_name ;
```

Summary

- Data definition sublanguage: CREATE, ALTER, DROP
- Simple queries: SELECT, FROM, WHERE, ORDER BY
- Set theoretic operations: IN, UNION, INTERSECT, EXCEPT, EXISTS
- Aggregate functions: AVG, MIN, MAX, COUNT; GROUP BY, HAVING
- Database monification: INSERT INTO, UPDATE, DELETE FROM
- Views and joints
- Embedded SQL, cursors