Workshop II: SQL

Authors: István Kiss's material was supplemented by Ferenc Unghváry

1. Introduction to the SQL

- Its development started around 1974-75 at IBM, under the original name SEQUEL (Structured English QUEry Language);
- From 1979 onwards SQL was present in the commercial products of several companies (such as IBM, ORACLE Corp.);
- ANSI standard since 1987.

Significance

- A standard language employed by almost every relational database management system (with some modifications);
- A compact, user friendly language suitable for database server and client communication;
- SQL is not a procedural programming language (at least in the queries).

Language definition

The present document presents the SQL dialect of the ORACLE database manager, which is very similar to other variations found in different products. The main focus is on SQL itself, therefore product or hardware specific elements of the language will be not / only partly covered.

The SQL language statements are sub-divided into several groups, including:
- Data Definition Statement (DDS);
- Data Manipulation Statements (DMS);
- Queries; and
- Data Control Statements (DCS).

Apart from the literals uppercase and lowercase letters are not differentiated (that is SQL is not case-sensitive). For the sake of better understanding SQL keywords are in all caps, and all other elements will be written in lower case.

Statements might spread across several lines; line-breaks have no semantic meaning. All SQL statements include the semicolon ("`;`") statement terminator.

1.1. Table definition statements

1.1.1. Creating tables

A new table can be added to the database using the

```
CREATE TABLE <table_name>
(<column_name1> <datatype1> [NOT NULL]
[, <column_name2> <datatype2> [NOT NULL] , ...]);
```

statements. The set of possible data types varies in different implementations, however the following base types are always there:

- `CHAR(n)` a text (string) containing a maximum number of n characters;
- `LONG` similar to `CHAR`, but there is no maximum size defined (extremely big);
- `NUMBER(n)` a maximum n characters long whole number with a sign (n represents the number of digits in the value);
- `DATE` date (and generally time).

In case a column contains the NOT NULL constraint, the corresponding field of the record must always contain a valid value.

The following example illustrates how to create some of the tables of the `SCOTT` demo database (with slight modifications):

```
CREATE TABLE customer (
   custid        NUMBER (6)   NOT NULL,
   name          CHAR (45),
   address       CHAR (40),
   city          CHAR (30),
   state         CHAR (2),
   zip           CHAR (9),
   area          NUMBER (3),
   phone         CHAR (9),
   repid         NUMBER (4),
   creditlimit   NUMBER (9,2),
   comments      LONG);
```

```
CREATE TABLE ord   (
   ordid         NUMBER (4)   NOT NULL,
   orderdate     DATE,
   commplan      CHAR (1),
   custid        NUMBER (6)   NOT NULL,
   shipdate      DATE,
   total         NUMBER (8,2));
CREATE TABLE item  (
   ordid         NUMBER (4)   NOT NULL,
   itemid        NUMBER (4)   NOT NULL,
   prodid        NUMBER (6),
   actualprice   NUMBER (8,2),
   qty           NUMBER (8),
   itemtot       NUMBER (8,2));
CREATE TABLE product (
   prodid        NUMBER (6),
   descrip       CHAR (30),
   partof        NUMBER (6),
   comments      LONG);
CREATE TABLE price (
   prodid        NUMBER (6)   NOT NULL,
   stdprice      NUMBER (8,2),
   minprice      NUMBER (8,2),
   startdate     DATE,
   enddate       DATE);
```

1.1.2. Deleting tables

To remove an entire SQL table the following

```
DROP TABLE <table_name>;
```

statement is used.

1.2. Data manipulation statements

1.2.1. Inserting data

Tables created with the `CREATE TABLE` statement are initially empty. To add a new row to an SQL table the following

```
INSERT INTO <table_name> [(<column_name1> [<column_name2>,
   ...])]
VALUES (<value1> [, <value2> , ...]);
```

statement is used.

In case the column list is not specified all fields must be assigned a value – according to the order of field names used in table creation – , otherwise only the fields corresponding to the column list entries must be assigned a value, all other fields will have the value of `NULL`.

Of course it might be that when inserting a new row the data added to a specified column has the value `NULL`, unless the column is specified with the `NOT_NULL` constraint.

Please note: the execution stops and results in an insert error when trying to add `NULL` value to a column against the constraint.

The insert statement can be used to add one row to a table at a time.

To insert two new products the product table the following 2 statements are used:

```
INSERT INTO product (prodid, descrip)
VALUES (111111, 'Steam engine');
INSERT INTO product
VALUES (111112, 'Oracle 6.0', NULL,' Relational DBMS');
```

1.2.2. Deleting data

One can delete one or more records from a given SQL table with the

```
DELETE FROM <table_name>
[WHERE <condition>]
```

statement.

If the `WHERE` clause is omitted, all rows in the table are removed. Otherwise any rows that match the `WHERE` condition will be removed from the table.

Let us consider our previous example: deleting the product having the ID (`prodid`) `111112` is done like this:

```
DELETE FROM product
WHERE prodid=111112;
```

1.2.3. Modifying data

Modifying one or more records in the table can be performed by the

```
UPDATE <table_name>
SET <column_name1> = <value1> [,<column_name2> = <value2>,
   ...]
[WHERE <condition>];
```

statement.

If the `WHERE` clause is omitted, all rows in the table are modified. Otherwise only the rows that match the `WHERE` condition will be modified.

Adding a comment to our product called `Steam engine` in table `product`:

```
UPDATE product
SET comment = 'designed by James Watt'
WHERE descrip=' Steam engine';
```

1.3. Queries

The general syntax of queries is the following:

```
SELECT <column_names>
FROM <table_names>
[WHERE <conditions>]
[<grouping>]
[<ordering>];
```

A `SELECT` statement returns a result-set of records from one or more tables which is stored in the result table – which might sometimes contain only one row and one column. The result table might be reused by being nested into other statements (e.g. set operations).

The `<column_names>` clause defines the result table columns;
the `<table_names>` clause defines the tables from which the results are selected;
the `<conditions>` optional clause specifies which rows to retrieve;

the <grouping> clause groups rows of the result table sharing a property;
the <ordering> specifies an order in which to return the rows.

The following chapters will illustrate how queries can be used to implement the primitive operations of relational algebra.

1.3.1. Projection (SELECT)

SELECT <column_names> FROM <table_name>;
Projection is an operation that returns the given columns from a table. The <column_names> clause is where the required columns are listed.
For example in case we would like to retrieve the product IDs and descriptions:

```
SELECT prodid, descrip FROM product;
```

Selecting all columns:

```
SELECT * FROM product;
```

In case the <column_names> clause is not specified and the * is used all table columns are selected. (The result table corresponds to the original table.)

Apart from the column names of the table specified in the <table_name> clause after the FROM keyword built-in SQL operations – e.g. simple arithmetic expressions creating new value, aggregate functions (see below) – can also be used in the <column_names> clause.
Calculating the price including the VAT (Value Added Tax) goes like this:

```
SELECT prodid, startdate, 1.25*stdprice FROM price;
```

The AS keyword (column synonym) can be used to give a separate name to 1.25*stdprice in the query, which can later serve as a reference in the ORDER BY (grouping) clause or - in case of nested queries - in the nesting context. For example:

```
SELECT prodid, startdate, 1.25*stdprice AS pricewithtax FROM
   price;
```

Based on the columns selected the result table might contain identical rows, which is contradicting one of the basic rules of relational tables. However, by default the SELECT statement does not filter out identical lines due to the time consuming nature of filtering. That is why the programmer must be aware of the potential errors caused by identical lines in the result tables. If applicable, identical lines can be eliminated by using the DISTINCT keyword.

Selecting all the different product descriptions:

```
SELECT DISTINCT descrip FROM product;
```

1.3.2. Restriction (WHERE)

A WHERE clause in SQL specifies that the SELECT statement should only affect rows that meet specified criteria. All rows for which the predicate in the WHERE clause is True are returned by the query.

The criteria are expressed in form of predicates. A predicate might contain the following elements:

literals for different data types: numbers, strings, dates;
column names;

expressions created using these elements and basic data operations

- by numbers: arithmetic operations (+, -, *, /), arithmetic functions;
- by strings: SUBSTR(), INSTR(), UPPER(), LOWER(), SOUNDEX(), ...;
- by dates: +,-, conversions; or
- sets e.g.: (10,20,30);

a whole SELECT statement within brackets (nested queries).

The expressions made from the data above are evaluated using the following operations:

- defining relations: <, <=, =, !=, >=, >;
- retrieving values in a range: BEEWEEN ... AND ...;
- checking for NULL: IS NULL, IS NOT NULL;
- checking values belonging to a set: IN <set>;
- checking strings:
- pattern matching ... LIKE <pattern>, where
  - % allows you to match any string of any length (including 0),
  - _ allows you to match on a single character.

Finally, predicates can be a combination of multiple predicates. The keywords AND, OR and NOT can be used to combine two predicates into a new one. If multiple combinations are applied, parenthesis can be used to group combinations.

Prices higher than 2000 USD:

```
SELECT prodid, startdate, stdprice FROM price
WHERE stdprice > 2000;
```

Prices (including the VAT) higher than 2000 USD in an increasing order:

```
SELECT prodid, startdate, 1.25*stdprice AS pricewithtax
FROM price
WHERE 1.25*stdprice > 2000
ORDER BY pricewithtax;
```

Prices valid on the 8th of March, 1994:

```
SELECT prodid, stdprice FROM price
WHERE '08-mar-94' BETWEEN startdate AND NVL(enddate,'31-dec-
   94');
```

The NVL(<column_name>,<value>) function lets you substitute a value when a null value is encountered - in the example if enddate is not specified (has a value of NULL) then it is substituted with 31-dec-94.

Prices lower than 2000 USD and minimal price is not specified:

```
SELECT prodid, startdate, stdprice FROM price
WHERE stdprice < 2000 AND minprice IS NULL;
```

1.3.3. Join

A join clause is used to combine rows from multiple tables, resulting in a new, temporary table, sometimes called a "joined table". An **inner join** requires each record in the two joined tables to have a matching record. The tables involved in the join are listed (separated by a comma) in the <table_names> clause after the FROM keyword, while the columns involved are defined in the <conditions> clause after the WHERE keyword of the SELECT statement.

The name, price and price validity of each product:

```
SELECT product.descrip, price.*
FROM product, price
WHERE product.prodid=price.prodid;
```

As it can be inferred from the example above, the names of the columns based on which the join is performed are the same (`prodid`); and the names of the joined tables are written before the column names in the `WHERE` clause (and are separated by a period).

The same situation might occur in the `<column_names>` clause following the `SELECT` keyword. As an example, the query showed above takes all the records from the Products table and finds the matching record(s) in the Price table, based on the join predicate. The join predicate compares the values in the Product ID column in both tables. If it finds no match, then the joined record remains outside the joined table, i.e., outside the result of the join. However, SQL provides means to make the joined table retain each record—even if no other matching record exists. This is called an **outer join**.

Let us consider the previous example with some modifications:

```
SELECT      product.descrip, price.stdprice, price.startdate
FROM product, price
WHERE       product.prodid=price.prodid(+);
```

Herein (+) marks the "left" table - this means that a left outer join returns all the values from the left table (`price`), plus matched values from the right table (`product`) (or `NULL` in case of no matching join predicate).

A table might be referenced more than one time in the `<conditions>` clause after the `WHERE` keyword.

Products with the same name (by pairs):

```
SELECT      a.descrip, a.prodid, b.prodid
FROM product a, product b
WHERE       a.descrip=b.descrip AND a.prodid<b.prodid;
```

A table that is referenced more times can be given a local name in the `<table_names>` clause after the `FROM` keyword. Local names can be used in case of different tables, as well.

Apart from the join clause other logical expressions might be simultaneously used.

1.3.4. Aggregate functions

Aggregate functions can be used to create one single value based on the values of the records in a given column of the result table. For example:

- `AVG()` aggregate function selects the average value for certain table column,
- `SUM()` aggregate function allows selecting the total for a numeric column,
- `COUNT()` aggregate function is used to count the number of rows in a database table,
- `MAX()` aggregate function allows us to select the highest value for a certain column,
- `MIN()` aggregate function allows us to select the lowest value for a certain column.

Average of prices valid from the 1st of January, 1994:

```
SELECT AVG(stdprice) FROM price WHERE startdate='01-jan-
   1994';
```

The total number of products:

```
SELECT COUNT(*) FROM product;
```

The total number of products with different names:

```
SELECT COUNT(DISTINCT descrip) FROM product;
```

Average minimum price:

```
SELECT AVG(NVL(minprice, stdprice)) FROM price;
```

In case the result table contains columns other than aggregate function results (and other than constants), then result table data must be grouped based on these columns. Grouping is required even when the programmer is certain that all selected rows have the same values in those particular columns.

The following is a valid statement (since both result columns are aggregate function results):

```
SELECT COUNT(*), AVG(stdprice) FROM price;
```

However, this statement is not correct (description might be different for the result table rows):

```
SELECT COUNT(*), descrip FROM product;
```

The following statement is not correct either, even though the `WHERE` condition guarantees that `startdate` has the same value (`01-jan-94`) in every resulting row:

```
SELECT startdate, AVG(stdprice) FROM price
WHERE startdate = '01-jan-94';
```

A correct solution would be:

```
SELECT startdate, AVG(stdprice) FROM price
WHERE startdate = '01-jan-94'
GROUP BY startdate;
```

1.3.5. Nested queries

In nested queries there might be another complete `SELECT` statement after the `WHERE` keyword.

List of products having a price valid from 1994:

```
SELECT prodid, descrip FROM product
WHERE prodid IN
    (SELECT prodid FROM price
     WHERE startdate >= '01-jan-94')
```

As it can be inferred from the above example first those product IDs are selected that have a starting date of 1994 or later, and secondly we examine every row in the product's table to find out if the ID of a particular product is in the result set of the nested query. The same result can be obtained using join, as well:

```
SELECT prodid, product.descrip
FROM product, price
WHERE product.prodid = price.prodid
AND price.startdate >= '01-jan-94';
```

However, please note that these two solutions are not equivalent: in case the price of the product changed in 1994 the query with the join generates a row in the result table for every price entry; thus opposing the nested query solution.

A nested query results in either one single value - because one column of a particular row is selected, or an aggregate function is used - or multiple values (multiple rows).

In the previous example the result value of the nested `SELECT` was used the same way as a basic value. If the `SELECT` provides a result set, set operations can be employed. Apart from the basic `IN()` operation `ANY()` and `ALL()` can be also used (the relation is True for at least one / all values of the set).

Products having the highest price (the might be more results):

```
SELECT prodid, stdprice FROM price
WHERE stdprice >= ALL (SELECT stdprice FROM price);
```

Same example as above, but this time with an aggregate function:

```
SELECT prodid, stdprice FROM price
WHERE stdprice = (SELECT MAX(stdprice) FROM price);
```

1.3.6. Grouping

Aggregate functions are run for every row in the result table. It might often be useful to group the selected rows based on a certain criteria and apply the aggregate functions on these groups instead of the whole table.
The highest price from the same day:

```
SELECT startdate, MAX(stdprice) FROM price
GROUP BY startdate;
```

Similar to the use of aggregate functions only the column name used for the grouping and the aggregate functions applied on these groups can be listed in the `<column_names>` clause after the `SELECT` keyword. After the grouping certain groups might be excluded from the result table.

Minimal prices on the same day within the 1000 USD and 3000 USD range:

```
SELECT startdate, MAX(stdprice) AS maxprice FROM price
GROUP BY startdate
HAVING maxprice BETWEEN 1000 AND 3000;
```

Please note that only mutual characteristics' values - columns used as grouping basis, results of aggregate functions - can be listed after the `HAVING` keyword. Of course `WHERE` conditions might be used before the grouping. It is recommended to use `WHERE` conditions whenever possible (it is faster), and only use the `HAVING` structure when values depending on the whole group are checked.

1.3.7. Ordering

In the result tables of all the queries discussed before the order of rows is random - and cannot be defined by the programmer. The order of the resulting rows can be controlled using the ordering specified after the `ORDER BY` keywords. It is possible to order by more than one column; the output will be ordered according to the first column. If there is a tie for the value of the first column, we then sort by the second column etc. For every column the direction of the ordering can be specified: `ASC` means that the results will be shown in ascending order, and `DESC` means that the results will be shown in descending order. If neither is specified, the default is `ASC`.

The price of product number 111111 (ordered by starting date):

```
SELECT stdprice, startdate FROM price WHERE prodid=111111
ORDER BY startdate;
```

The highest price ever for each product in a descending order:

```
SELECT prodid, MAX(stdprice) FROM price
GROUP BY prodid
ORDER BY MAX(stdprice) DESC;
```

1.3.8. Set operations

Result tables created by queries can be considered as sets, which can serve as a basis to perform set operations provided by SQL. The set operations are:

`UNION` – combines the results of two queries together;

`INTERSECT` – value is selected only if it appears in both result tables;

`MINUS` – takes all the results from the first SQL statement, and then subtract out the ones that are present in the second SQL statement;

`IN` – we know exactly the value of the returned values we want to see for at least one of the columns.

Set operations must be written between two `SELECT` statements. (See the previous chapter about nested queries!)

1.3.9. Hierarchical relationship query

Relational tables can also be used to describe hierarchical relationship between data rows.
For example the following statement:

```
SELECT descrip, prodid, partof
FROM product
CONNECT BY PRIOR prodid = partof
START WITH descrip = 'Steam engine';
```

lists all the components of the steam engine until the last tiny screw.

The record-pairs meeting the condition specified in the `CONNECT BY` clause are in a parent-child relationship in the hierarchy. The statement marked with the keyword `PRIOR` refers to the parent record. Thus, while `PRIOR prodid` might be the ID of the piston, the integration of the piston valve into the system is described by the `partof` attribute in the above example.

1.3.10. Other related non-SQL statements

Most systems also contain not strictly SQL-related statements that are used to set result table display attributes - such as column names, column width, data format, alignment. For more information on this topic please refer to the Server SQL Language Reference Manual help.

1.4. Views

Views are virtual tables which feature different logical model and grouping of the data stored using the physical tables. Views can be created using the following statement:

```
CREATE VIEW <view_name> [(<column_name1> [, <column_name2>,
    ...])]
AS <query>;
```

The only limitation of the query is that it might not contain ordering. In the case column names are not specified the columns of the view are identical to the columns listed in the `<column_names>` clause after the `SELECT` keyword. However, the view column names must be specified if the query uses aggregate functions.

For example the following view shows the actual price of each product:

```
CREATE VIEW prods AS
   SELECT product.prodid, product.descrip pdescrip,
     x.stdprice sprice, x.minprice mprice
   FROM product, price x
   WHERE x.prodid = product.prodid
   AND x.startdate >= all (
     SELECT startdate
     FROM price i
     WHERE i.prodid = x.prodid);
```

Views can be used the same way as a table in queries. The main importance of views lies in their ability to create another data model, to hide parts of the information – e.g. different users can analyze the data through different views. A view is usually read-only, and might only appear in data modification operations when it was created from a single table and does not contain aggregated value. Note, that a view always shows up-to-date data. The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

A view can be deleted using the

```
DROP VIEW <view_name>;
```

statement. Therefore the view created in the example above can be deleted like this:

```
DROP VIEW prods;
```

1.5. Indexes

An index can be created in a table to find data more quickly and efficiently.

1.5.1. Creating indexes

An index can be created using the following:

```
CREATE [UNIQUE] INDEX <index_name>
ON <table_name> (<column_name1> [, <column_name2> , ...]);
```

statement.

Indexes are updated by the database manager upon every table modification. In case the index was created with the `UNIQUE` keyword the system ensures that all fields in the given column have a different value. Complex indexes combining more columns can also be created. After their creation indexes are invisible to the user but their usage results in increased query speeds. A user should only create indexes on columns (and tables) that will be frequently searched against. Without an index the whole table must be read for every query. However, in case there is an appropriate index created only the requested rows will be read from the disk by the database manager.

For example the following statement:

```
SELECT * FROM emp WHERE ename = 'JONES';
```

returns Jones' record from table `emp` without looking for it if there is an index created for the `ename` column. Indexes make queries quicker and more efficient, even if they belong to only one of the search conditions.

1.5.2. Deleting indexes

Indexes can be deleted using the

```
DROP INDEX <index_name>
```

statement.

1.6. Assigning privileges

The SQL command GRANT allows to assign system privileges and object privileges to users and roles. The syntax of the GRANT statement is the following:

```
GRANT [DBA | CONNECT | RESOURCES]
TO <username1> [, <username2> , ...]
IDENTIFIED BY <password1> [, <password2>, ...];
```

The DBA privilege defines the database administrators (DataBase Administrator) who have unlimited access to all database objects - they can create, delete and modify anything, and also control other object storage and access parameters. New objects can be created, modified and deleted by all users with RESOURCES privilege. CONNECT only enables the user to logon into the database.

Access to particular objects - tables or views - can be set by the statement below:

```
GRANT <privilege1> [, <privilege2> , ...]
ON <table_name> | <view_name>
TO <username>
[WITH GRANT OPTION];
```

In the <privileges> clause the privileges can be defined. The possible privileges are:

```
ALL,
SELECT,
INSERT,
UPDATE <column_name>, ...
DELETE,
ALTER,
INDEX
```

The last two privileges cannot be assigned to views. The key word PUBLIC indicates that the privileges are to be granted to all users, including those that may be created later. The WITH GRANT OPTION keywords convey the privilege or role to <username> with the right to grant the same privileges or role to other users.

1.7. Modifying table definitions

Use the ALTER TABLE statement to modify the definition of an existing table. The syntax is:

```
ALTER TABLE <table_name>
[ADD | MODIFY] <column_name> <data_type>;
```

where ADD adds a new, NULL value column to the table while MODIFY is used to change the width of an existing column.

For example adding a new column to table mytable:

```
ALTER TABLE mytable
ADD id NUMBER(6);
```

1.8. Transactions

Usually, database modifications cannot be performed in one step only, since the modification generally involves changing information stored in multiple tables, or in multiple rows; or inserting more records. Moreover, there might be situations where the user changes his mind halfway through the modification or the database manager stops resulting in even more

serious consequences. Since some parts of the desired modifications are already done and some aren't data may become inconsistent.

A transaction is a collection of one or more SQL statements that is treated as a single unit of work. If one statement in a transaction fails, the entire transaction can be rolled back (canceled). If the transaction is successful, the work is committed and all changes to the database from the transaction are accepted.

An in-process transaction can be finished with the COMMIT statement, which finalizes all modifications since the previous COMMIT; or the ROLLBACK statement can be used to cancel these in-between modifications to return to the valid state of the previous COMMIT.

The automatic commit property of SQL operations can be set using the following statement:

```
SET AUTOCOMMIT [ON | OFF];
```

When automatic commit is ON, the successful execution of all SQL statements also means a COMMIT. When auto-commit is OFF only ALTER, CREATE, DROP, GRANT and EXIT statements are committed (meaning these statements cannot be rolled back - e.g. there is no undo for table deletion).

Upon restarting from a hardware error, or an INSERT, UPDATE or DELETE statement error the system automatically performs a ROLLBACK. In order to avoid the rollback caused by a possible statement error it is highly recommended to use the COMMIT statement in a safe state.

1.9. Controlling concurrent accesses

Database management systems are generally used by multiple users at a time, which sometimes might cause some difficulties. Therefore parallel access to tables can be controlled separately:

```
LOCK TABLE <table_name1> [, <table_name2> , ...]
IN [SHARE | SHARED UPDATE | EXCLUSIVE] MODE [NOWAIT];
```

The LOCK statement can be used by a user to define the parallel access right of other users to the given tables. When executing a LOCK statement the system checks whether the access mode requested by the LOCK statement is compatible with the lock currently in effect, or not. If it is compatible, the statement returns and the user can execute the next statement. If the required lock is not available the statement waits until the lock currently in effect is released (when the statement does not contain the NOWAIT keyword). Without NOWAIT the statement returns and might issue an error message.

Table access is defined by the first successful LOCK statement.

After the LOCK TABLE statement with the IN EXCLUSIVE MODE option executes successfully, no other user can obtain a lock on the specified table. The SHARE keyword locks a table in shared mode which means other processes cannot update or delete data. The SHARE UPDATE keyword means other can update data - herein mutual exclusion on row-level is automatically provided by ORACLE - but not the same row at the same time.

1.10. Constraints

In the table definitions so far only the column names and data types, and the NOT NULL constraints were defined. However it might be useful to assign stricter constraints to tables, to define conditions that are verified by the system upon every modification (for more

information on this topic please refer to the Appendix entitled "Database constraints in Oracle"). For example:

- more precise definition of data co-domains (e.g. being part of an interval or a subset, where the set itself is consisted of values from another table column);
- a column being primary key, meaning it has a different value for every record (similar effect can be achieved by using `UNIQUE` indexes);
- a column being foreign key, meaning that it matches one of the values of another table's primary key columns.

In case the constraints are violated during table modification the system generates an exception and runs an exception handling routine, if applicable.
For more SQL elements please refer to the description in the on-line help.