SQL (-86) deel 2 over:

DDL, Insert/Update/Delete, Commit/Rollback, Grant/Revoke e.d.

0 Inhoud van dit deel van het IS0-dictaat:

1 Creating and Manipulating Table Definitions ( Create, Alter+Add, Drop )
   1.1 CREATE TABLE
   1.2 DROP TABLE
   1.3 ALTER TABLE
   1.4 CREATE [UNIQUE] INDEX
   1.5 Use of indexes for performance enhancement
   1.6 Use of indexes as a data integrity check
   1.7 DROP INDEX

2 Updating the database ( Insert, Update+Set, Delete )
   2.1 Adding rows to a table: the INSERT command
   2.2 Updating rows of a table: the UPDATE/SET-command
   2.3 Deleting rows from a table: the DELETE-command

3 Controlling the Execution of Commands ( Commit / Rollback Work )
   3.1 AUTOCOMMIT modes
   3.2 The ROLLBACK WORK command

4 Granting and Revoking User Privileges
   4.1 GRANT
   4.2 REVOKE

We laten hierna de uitwerking van de verschillende onderdelen volgen.

0.1 Inleiding op dit tweede SQL-deel van de IS0-reader
Ook dit deel van de IS0-reader is gebaseerd op een engelstalig ‘manual’ bij een nogal verouderd
SQL-database-systeem, dat was gebaseerd op de SQL-1-versie (uit 1986).
In dit deel van de IS0-reader gaan we vooral in op de DDL-mogelijkheden van SQL-1 en daarnaast op
commando’s voor het manipuleren van gegevens (toevoegen, veranderen, verwijderen) in de database
en op de mogelijkheden om onbevoegd gebruik van de database tegen te gaan.

1 Creating and Manipulating Table Definitions ( Create, Alter+Add, Drop )
The first stage in setting up an SQL database is to define the tables. Tables can only be created by
users with DBA or RESOURCE authority.

1.1 CREATE TABLE
The syntax of the CREATE TABLE command is:

CREATE TABLE table-name
   (column-name-1 data-type-1  [ NOT NULL ]
    [, column-name-2 data-type-2  [ NOT NULL ] ...]
   [ IN dbspace-name ]


table-name, column-name:
consist of up to 18 characters which can be upper or lower case letters, numbers, $, #, @ or underscore. The first character cannot be a number.

data-type:
may be any of the following:
1. SMALLINT  A whole number (integer) between \(-32,767\) and \(+32,767\).
2. INTEGER  A whole number (integer) between \(-2,147,483,647\) and \(+2,147,483,647\).
3. DECIMAL \((m, n)\)  A decimal number consisting of \(m\) digits (maximum 18), an optional decimal marker located to the left of \(n\) digits and an optional sign (+ or -). In other words, a total of \(m\) digits, \(n\) of those after the decimal point and \(m-n\) before it, possibly with a numeric sign.
4. FLOAT  A floating point number. For example: 1, 1.5, 1.23E45, 1.23E-45. The maximum floating point value depends on the actual computer hardware being used.
5. CHAR \((n)\)  A character string of fixed length \((n)\) where \(n\) is at least 1 and at most 254.
6. VARCHAR \((n)\)  A variable length character string of maximum length \((n)\) where \(n\) is at least 1 and at most 254. SQL will only store the actual length.
7. LONG VARCHAR  A variable length character string of up to 32,767 characters.

Note:
Some RDBMS may have additional data-types like DATE, TIME, and BINARY etc.

NOT NULL
specifies that the column cannot contain any null fields. In other words, each field of the column must always contain a value.

dbspace-name
is the name of the database file in which the table is to be placed. If the IN dbspace-name clause is omitted, the table will be placed according to the following rules:
• If you own one or more private dbspaces, the table will be created in the private dbspace you acquired first.
• If you do not own any private dbspaces and you have DBA authority, the table will be created in the main dbspace. This is the dbspace that is created when the database is initially created. Its name is the same as that of the database.

1.1.1 Example
Create a table which can hold all the facts about presidents as presented in the table PRESIDENT (zie het eerste reader-deel over SQL van dit collegedictaat):

```
CREATE TABLE PRESIDENT
  ( PRES_NAME CHAR (15) NOT NULL,
    BIRTH_YR SMALLINT NOT NULL,
    YRS_SERV SMALLINT NOT NULL,
    DEATH_AGE SMALLINT,
    PARTY CHAR (12) NOT NULL,
    STATE_BORN VARCHAR (14) NOT NULL )
```
1.2 DROP TABLE
The syntax of the DROP TABLE command is as follows:

```
DROP TABLE table-name
```

When a table is dropped, the table definition, contents, indexes and views on the table, as well as all permissions to the table, are also dropped.

1.3 ALTER TABLE
A table definition can be modified to include new columns. The ALTER TABLE command appends a new column to the right hand side of an existing table. A column can be added irrespective of whether the table already contains rows of information, or is empty.
The syntax of the ALTER TABLE command is:

```
ALTER TABLE table-name ADD column-name data-type
```

The arguments table-name, column-name, and data-type are the same as for the CREATE TABLE command.

1.3.1 Example
Add a new column to the table STATE for the population of the state at the time of entry to the union of states.

```
ALTER TABLE STATE ADD POPENTERED SMALLINT
```

To see the effect of the ALTER TABLE command, suppose the query to display all states beginning with ’M’ is issued.

```
SELECT * FROM STATE
WHERE STATE_NAME LIKE 'M%
```

Result:

<table>
<thead>
<tr>
<th>STATE_NAME</th>
<th>ADMIN_ENTERED</th>
<th>YEAR_ENTERED</th>
<th>POPENTERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Massachusetts</td>
<td>?</td>
<td>1776</td>
<td>?</td>
</tr>
<tr>
<td>Maryland</td>
<td>?</td>
<td>1776</td>
<td>?</td>
</tr>
<tr>
<td>Mississippi</td>
<td>8</td>
<td>1817</td>
<td>?</td>
</tr>
<tr>
<td>Maine</td>
<td>8</td>
<td>1820</td>
<td>?</td>
</tr>
<tr>
<td>Missouri</td>
<td>9</td>
<td>1831</td>
<td>?</td>
</tr>
<tr>
<td>Michigan</td>
<td>12</td>
<td>1837</td>
<td>?</td>
</tr>
<tr>
<td>Minnesota</td>
<td>18</td>
<td>1858</td>
<td>?</td>
</tr>
<tr>
<td>Montana</td>
<td>26</td>
<td>1889</td>
<td>?</td>
</tr>
</tbody>
</table>

Note:
- A column cannot be inserted amongst columns that have already been defined for a table. It can only be appended to the right hand side of the table.
- A column that is added to a table cannot be specified as NOT NULL, since there is no way of specifying a value for the column for each of the rows that might already exist in the table.
1.4 CREATE INDEX
Creating indexes for your tables is an extremely important part of setting up a database. Indexes can dramatically improve the performance of SQL queries, and also provide automatic checking for unwanted duplication of data.
Indexes provide a quick means of finding rows in a table. All SQL commands which have a WHERE clause use indexes if possible. This means that indexes can improve the performance of the SELECT, UPDATE and DELETE commands.

The use of indexes by SQL is invisible to the user. SQL knows which indexes are available and automatically determines the most efficient combination of indexes to answer your query. This is called automatic optimization.
Indexes can be created or dropped at any time during a table’s lifetime. If a new index has been created to speed up the evaluation of a query, that index is used automatically in later processing of that query. Furthermore, if an index is dropped, queries that previously used the index will use an alternative method to obtain the necessary result. In other words, routines, stored queries and commands entered interactively are all independent of the existence of indexes.

The syntax of the CREATE INDEX command is:

```
CREATE [UNIQUE] INDEX index-name ON [creator.]table-name
    (column-name-1 [ASC | DESC], column-name-2 [ASC | DESC] ...)
```

**UNIQUE**
specifies that each field value (or combination of field values) for the columns specified in the CREATE INDEX command may appear only once in the table.

**Note:** Within a unique index, NULL is considered to be a value. This means that NULL can occur only once in a column which has a unique index defined on it.

**ASC | DESC**
specifies the ordering of the rows used for the internal implementation of the index.
ASC indexes are sometimes more efficient.

A table can have many indexes, and each index can reference up to 16 columns in the table.
An index can only apply to a single table.

Two examples of indexes are as follows:

```
CREATE INDEX STATE_NAME_INDEX ON STATE (STATE_NAME)
CREATE INDEX STATE_ADMIN_INDEX ON STATE (STATE_NAME, ADMIN_ENTERED)
```

We can represent these two indexes diagrammatically as follows:

```
STATE_NAME | ADMIN_ENTERED | YEAR_ENTERED
< = = = = = = = = = >             STATE_NAME_INDEX
< = = = = = = = = = = = = = = = = = = = = >         STATE_ADMIN_INDEX
```

1.5 Use of indexes for performance enhancement
To make the best use of indexes, you need to understand when they are used. The following SELECT command uses the index STATE_NAME_INDEX to produce the SELECT result of:
SELECT *  
FROM STATE  
WHERE STATE_NAME = 'Texas' 

Instead of having to scan the entire STATE table looking for rows where the state is Texas, SQL can use the index to go directly to the correct row.

The following WHERE clauses would also use STATE_NAME_INDEX:
  WHERE STATE_NAME > 'Idaho'  
  WHERE STATE_NAME < 'Utah'  
  WHERE STATE_NAME IN ('Kansas', Oregon', 'Nevada')  
  WHERE STATE_NAME BETWEEN 'Idaho' AND 'Utah'

The LIKE operator is the only operator that never uses an index. For example, the following WHERE clause would not use STATE_NAME_INDEX:
  WHERE STATE_NAME LIKE 'M%'

However, in this case the restriction can be overcome by using the BETWEEN operator. The following WHERE clause could be used instead of the one containing the LIKE clause:
  WHERE STATE_NAME BETWEEN 'M' AND 'MZ'

Multi-column indexes like STATE_ADMIN_INDEX are used to go directly to the rows which satisfy the WHERE clause if all columns covered by the index are used in the WHERE clause, and the comparisons are connected by AND.

For example, the following WHERE conditions use STATE_ADMIN_INDEX to find the correct rows:
  WHERE STATE_NAME > 'Idaho' AND ADMIN.Entered < 10  
  WHERE STATE_NAME BETWEEN 'Idaho' AND 'Utah'  
    AND ADMIN.Entered BETWEEN 10 AND 20

The rows which satisfy the WHERE clauses given above are obtained directly using the index STATE_ADMIN_INDEX. SQL does not have to scan any rows to determine whether they satisfy the conditions.

The exception is when one or more of the columns covered by the index is compared to a set of values. In other words, if 'IN, =ANY, =ALL or another set operation has been used, a multi-column index is used for one column only.

The following WHERE-condition does not use STATE_ADMIN_INDEX:
  WHERE STATE_NAME IN ('Idaho', 'Utah')  
    AND ADMIN.Entered BETWEEN 10 AND 20

The WHERE clause above would use STATE_NAME_INDEX to help find all the state names in the specified set, and scan these for the ADMIN.ENTERED condition.

Multi-column indexes can also be used to answer queries if the WHERE clause refers to only the first column in the index, and there are no other indexes on this column which could be used to answer the query more quickly. This is less efficient than using an index which is defined only on the required column, but is usually more efficient than no index at all. If there is an index defined on the single column needed to answer the query, it will be used in preference to a multi-column index.

In our example, STATE_NAME_INDEX would be used to answer the queries on STATE_NAME, except when ADMIN.ENTERED is used with an AND, when STATE_ADMIN_INDEX is preferred. This is called major key processing. On the other hand, STATE_ADMIN_INDEX cannot be used to answer queries which only refer to ADMIN.ENTERED in their WHERE clause. Major key processing only works for the first columns in the index.
Indexes are aids solely to improve the performance of SQL in retrieving table rows, and are not necessary for accessing stored data. If no appropriate indexes can be found on a table when processing a particular SELECT command, the entire table will be scanned to find the rows which satisfy the conditions. This is much slower than using an index if the result is only a small proportion of the rows in a large table.

Indexes speed up retrieval of data but slow down operations involving inserting new data or updating existing data because of the overhead in maintaining the index. For example, when a new row is inserted into a table, indexes for that table have to be updated to include the new row.

You should think carefully about the kinds of queries that you expect to be asked most often, and define indexes which help to answer these queries.

1.6 Use of indexes as a data integrity check
In addition to speeding up evaluation of your queries, unique indexes provide a valuable means of checking the integrity of information entered in a table. SQL will not accept duplicates in columns on which a unique index is defined. For instance, SQL will not allow you to add an additional row to the STATE table with the name Texas if the unique index STATE_NAME_INDEX exists. SQL also prevents a row in the table from being changed so that there are duplicates in the columns which are covered by a unique index. This means that you do not have to worry about unwanted and possibly disastrous duplicate information being entered.

1.7 DROP INDEX
The syntax for the DROP INDEX command is: DROP INDEX index-name
2 Updating the database (Insert, Update+Set, Delete)

There are three methods of update available:
   A) Add one or more new rows to a table within the database
   B) Modify one or more existing rows of a table within the database
   C) Delete one or more existing rows from a table within the database

2.1 Adding rows to a table: the INSERT command

The INSERT command has two formats:
   Format 1 adds a single new row to a table.
   Format 2 adds one or more existing rows (which are selected or computed from other tables) into a table.

The two formats are described below.

Format 1

```
INSERT [ INTO ] [ creator. ] { table-name | view-name } [ ( column-names ) ]
VALUES ( data-items )
```

Data-items is a list of one or more values, separated by commas, to be inserted into the new row for the columns specified in column-names. The first data-item value is inserted in the first column-name specified; the second data-item value is inserted in the second column-name specified, etc. A null value is inserted into a field by typing the word NULL for the data-item value. The data items must be compatible in type to the matching columns. Column names from the table that were not specified in the column-name list are given a null value; therefore, you must remember to include in the list all the NOT NULL columns in the table.

If no column names are specified, then a column list in the order of the table template is assumed.

Format 2

```
INSERT INTO [ creator.] { table-name | view-name } [ ( column-names ) ]
select-statement
```

The select statement selects the data to be inserted from another table. The number of columns retrieved by the select statement must match the number of columns for the insert. The columns selected must be type-convertible with the columns into which they will be inserted (numeric to numeric, character to character).

Notes:
1. Creator is not necessary for tables created by the current user.
2. Single quotes are required around character data values. A quote is represented in a string by two consecutive quotes.
3. The table specified in the FROM clause of the Select statement must not be the same as the table into which information is being inserted. You cannot use the INSERT command to duplicate rows or parts of rows in a single table.

Example:
To add a new row to the VERY_RECENT_PRES table (described in a previous chapter):

```
INSERT INTO VERY_RECENT_PRES
VALUES ( 'Nextpres I M' , 1940, NULL, 'Teaparty' , 'Allstates' )
```

If we now enter the query:
```
SELECT *
FROM VERY_RECENT_PRES
```
Result:

<table>
<thead>
<tr>
<th>PRES_NAME</th>
<th>BIRTH_YR</th>
<th>YRS_SERV</th>
<th>DEATH_AGE</th>
<th>PARTY</th>
<th>STATE_BORN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carter J E</td>
<td>1924</td>
<td>4</td>
<td>?</td>
<td>Democratic</td>
<td>Georgia</td>
</tr>
<tr>
<td>Ford G R</td>
<td>1913</td>
<td>2</td>
<td>?</td>
<td>Republican</td>
<td>Nebraska</td>
</tr>
<tr>
<td>Johnson L B</td>
<td>1908</td>
<td>5</td>
<td>65</td>
<td>Democratic</td>
<td>Texas</td>
</tr>
<tr>
<td>Kennedy J F</td>
<td>1917</td>
<td>2</td>
<td>46</td>
<td>Democratic</td>
<td>Massachusetts</td>
</tr>
<tr>
<td>Nixon R M</td>
<td>1913</td>
<td>5</td>
<td>?</td>
<td>Republican</td>
<td>California</td>
</tr>
<tr>
<td>Reagan R</td>
<td>1911</td>
<td>3</td>
<td>?</td>
<td>Republican</td>
<td>Illinois</td>
</tr>
<tr>
<td>Nextpres I M</td>
<td>1940</td>
<td>2</td>
<td>?</td>
<td>Teaparty</td>
<td>Allstates</td>
</tr>
</tbody>
</table>

Note that NULL has been entered to indicate the president has no death age.

Example 2:

```
INSERT INTO VERY_RECENT_PRES (PRES_NAME, BIRTH_YR)
SELECT CANDIDATE, ELECTION_YEAR
FROM ELECTION
WHERE WINNER_LOSER_INDIC = 'W'
AND ELECTION_YEAR > 1990
```

After this command has been processed, SQL can display a message like:

ARI5001 SQL PROCESSING SUCCESSFUL.
SQLCODE = 0 ROWCOUNT = 2.

The table VERY_RECENT_PRES will now contain:

<table>
<thead>
<tr>
<th>PRES_NAME</th>
<th>BIRTH_YR</th>
<th>YRS_SERV</th>
<th>DEATH_AGE</th>
<th>PARTY</th>
<th>STATE_BORN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carter J E</td>
<td>1924</td>
<td>4</td>
<td>?</td>
<td>Democratic</td>
<td>Georgia</td>
</tr>
<tr>
<td>Ford G R</td>
<td>1913</td>
<td>2</td>
<td>?</td>
<td>Republican</td>
<td>Nebraska</td>
</tr>
<tr>
<td>Johnson L B</td>
<td>1908</td>
<td>5</td>
<td>65</td>
<td>Democratic</td>
<td>Texas</td>
</tr>
<tr>
<td>Kennedy J F</td>
<td>1917</td>
<td>2</td>
<td>46</td>
<td>Democratic</td>
<td>Massachusetts</td>
</tr>
<tr>
<td>Nixon R M</td>
<td>1913</td>
<td>5</td>
<td>?</td>
<td>Republican</td>
<td>California</td>
</tr>
<tr>
<td>Reagan R</td>
<td>1911</td>
<td>3</td>
<td>?</td>
<td>Republican</td>
<td>Illinois</td>
</tr>
<tr>
<td>Nextpres I M</td>
<td>1940</td>
<td>2</td>
<td>?</td>
<td>Teaparty</td>
<td>Allstates</td>
</tr>
</tbody>
</table>

2.2 Updating rows of a table: the UPDATE/SET .. -command

The syntax of the UPDATE command is:

```
UPDATE [creator.] table-name
SET     column-name-1 = expression-1
        [ , column-name-2 = expression-2 ] . . .
[ WHERE  search-condition ]
```

column-name-1
is the name of the first column to be updated.

d expression-1
is the new value to be placed in column-name-1. This expression may be any SQL expression, including a subquery. It may contain constants, NULL, column names and the arithmetic operators +, -, *, /.

column-name-2
is the name of the second column to be updated.
expression-2

is the new value to be placed in column-name-2.

search-condition

specifies the rows to be updated (same power as in the SELECT command).

Note:

All rows that satisfy the specified condition are updated; if no search condition is given, all rows in the named table are updated.

Example

\[
\text{UPDATE } \text{VERY}_\text{RECENT}_\text{PRES} \\
\text{SET} \quad \text{STATE}\_\text{BORN} = \text{\textquote{Texas}} \\
\text{WHERE} \quad \text{STATE}\_\text{BORN} \text{ IS NULL}
\]

Result (shown after submitting a Select command):

<table>
<thead>
<tr>
<th>PRES_NAME</th>
<th>BIRTH_YR</th>
<th>YRS_SERV</th>
<th>DEATH_AGE</th>
<th>PARTY</th>
<th>STATE_BORN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kennedy J F</td>
<td>1917</td>
<td>2</td>
<td>46</td>
<td>Democratic</td>
<td>Massachusetts</td>
</tr>
<tr>
<td>Johnson LB</td>
<td>1908</td>
<td>5</td>
<td>65</td>
<td>Democratic</td>
<td>Texas</td>
</tr>
<tr>
<td>Nixon R M</td>
<td>1913</td>
<td>5</td>
<td>?</td>
<td>Republican</td>
<td>California</td>
</tr>
<tr>
<td>Ford G R</td>
<td>1913</td>
<td>2</td>
<td>?</td>
<td>Republican</td>
<td>Nebraska</td>
</tr>
<tr>
<td>Carter J E</td>
<td>1924</td>
<td>4</td>
<td>?</td>
<td>Democratic</td>
<td>Georgia</td>
</tr>
<tr>
<td>Reagan R</td>
<td>1911</td>
<td>3</td>
<td>?</td>
<td>Republican</td>
<td>Illinois</td>
</tr>
<tr>
<td>Nextpres I M</td>
<td>1940</td>
<td>2</td>
<td>?</td>
<td>Teaparty</td>
<td>Allstates</td>
</tr>
</tbody>
</table>

The values in another table can be used to update the rows of a particular table. To do this, a subquery is used to set new column values.

Example

Set the party of all presidents whose party is unknown to the party which was in power when the state in which the president was born entered the union.

\[
\text{UPDATE } \text{RECENT}\_\text{PRESIDENTS} \\
\text{SET } \text{PARTY} = ( \text{SELECT } \text{PARTY} \\
\quad \text{FROM } \text{PRESIDENT} \\
\quad \text{WHERE } \text{PRES}\_\text{NAME} = \\
\quad \quad ( \text{SELECT } \text{MIN}\left( \text{PRES}\_\text{NAME}\right) \\
\quad \quad \text{FROM } \text{ADMINISTRATION} \\
\quad \quad \text{WHERE } \text{ADMIN}\_\text{NR} = \\
\quad \quad \quad ( \text{SELECT } \text{ADMIN}\_\text{ENTERED} \\
\quad \quad \quad \text{FROM } \text{STATE} \\
\quad \quad \quad \text{WHERE } \text{STATE}\_\text{NAME} = \\
\quad \quad \quad \text{RECENT}\_\text{PRESIDENTS}\.\text{STATE}\_\text{BORN} ) ) ) \\
\text{WHERE } \text{PARTY} \text{ IS NULL}
\]

Result:

<table>
<thead>
<tr>
<th>PRES_NAME</th>
<th>BIRTH_YR</th>
<th>YRS_SERV</th>
<th>DEATH_AGE</th>
<th>PARTY</th>
<th>STATE_BORN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kennedy J F</td>
<td>1917</td>
<td>2</td>
<td>46</td>
<td>Democratic</td>
<td>Massachusetts</td>
</tr>
<tr>
<td>Johnson LB</td>
<td>1908</td>
<td>5</td>
<td>65</td>
<td>Democratic</td>
<td>Texas</td>
</tr>
<tr>
<td>Nixon R M</td>
<td>1913</td>
<td>5</td>
<td>?</td>
<td>Republican</td>
<td>California</td>
</tr>
<tr>
<td>Ford G R</td>
<td>1913</td>
<td>2</td>
<td>?</td>
<td>Republican</td>
<td>Nebraska</td>
</tr>
<tr>
<td>Carter J E</td>
<td>1924</td>
<td>4</td>
<td>?</td>
<td>Democratic</td>
<td>Georgia</td>
</tr>
</tbody>
</table>
Notes
- There must be only one column or expression in the subquery SELECT list. If you want to set several columns in a table to the values occurring in other tables, you will need a subquery for each column.
- The subquery must return a single value or none. If it returns no value, NULL is taken.
- The subquery is usually a correlated subquery. There is almost always a WHERE clause in the subquery. This is because you need to specify which rows from the source table are to be used to update which rows in the target table.
- If you only wish to update some rows of the target table, there should be a WHERE clause on the UPDATE command. In the example above, we only wished to update the rows for which we did not already know the party. The rows for which the party was already known were skipped by using the WHERE clause of the UPDATE command.

2.3 Deleting rows from a table: the DELETE -command
The DELETE command deletes all the rows of a named table which satisfy a specified search condition.
The syntax of the DELETE command is:

```
DELETE [ FROM ] [ creator. ] table-name
[ WHERE search-condition ]
```

`search-condition`
specifies the row or rows to be deleted (same power as in the SELECT command).

Example:
In the previous examples for the INSERT command (employing Format 1), the president Nextpres I M was added to the VERY_RECENT_PRES table.
In example 2 for the INSERT command (employing Format 2), the two fictitious presidents ’Honour I’ and ’Camelot M’ were added to the VERY_RECENT_PRES table.
To delete the rows that were inserted in the above examples, the following DELETE statement could be used:

```
DELETE FROM VERY_RECENT_PRES
WHERE BIRTH_YR >= 1940
```

Note
INSERT and UPDATE should not be confused with ALTER TABLE (see previous chapter) which adds one new column on the right hand side of an existing table.
3 Controlling the Execution of Commands (Commit / Rollback Work)

A command can be prevented from executing immediately, it can be started, cancelled or ended. There are a number of commands to be used for these and other purposes.

3.1 AUTOCOMMIT modes

Two modes of executing SQL commands exist:

- Mode AUTOCOMMIT ON:
  Every command that is entered commits the previously entered command. This means that every command is a transaction, and that we also have the option of entering ROLLBACK WORK to undo the last command.

- Mode AUTOCOMMIT OFF:
  Modifications of tables specified in a single SQL command, or in a series of consecutive SQL commands, are committed only upon encountering a COMMIT WORK command. This mode allows us to determine a logical unit of work, consisting, in general, of a series of SQL commands, which are executed as one transaction and are delimited by the command COMMIT WORK and ROLLBACK WORK (or CANCEL).

This mode can be specified using the SET AUTOCOMMIT command:

```
SET AUTOCOMmit { ON | OFF }
```

When the system is started, (generally) AUTOCOMMIT is ON by default.
If we detect an error in a command or logical unit of work, we may use the SQL commands ROLLBACK WORK or CANCEL to undo the current logical unit of work.

3.2 The ROLLBACK WORK command

ROLLBACK WORK ends a logical unit of work. If AUTOCOMMIT is OFF, all tables updated in the current logical unit of work will be restored to their appearance before the last COMMIT WORK command, or, if there was no such command, since the last AUTOCOMMIT SET OFF command. If AUTOCOMMIT is ON, and if we enter a ROLLBACK WORK immediately following an INSERT, DELETE or UPDATE command this table will be restored to its state prior to that command.
4 Granting and Revoking User Privileges
User privileges can be granted and revoked using the GRANT and the REVOKE commands respectively.

4.1 GRANT
GRANT is an SQL command which allows a user holding specific privilege(s), to grant them to one or more other users.

There are the following two formats for the GRANT command:

Format 1  For granting privileges on tables and views.
Format 2  For granting SPECIAL privileges.

Format 1 of the GRANT command is used to grant privileges on a table or view to other users. When using format 1 of the GRANT command, we must not only possess the privilege we want to grant to another user, but also the GRANT OPTION as well; the GRANT OPTION may or may not be passed along with the privilege.

The syntax of the GRANT command is as follows:

**Format 1** (For granting privileges on tables and views)

```
GRANT { list-of-privileges | ALL }
ON [creator.] { table-name | view-name }
TO { list-of-users | PUBLIC }
[ WITH GRANT OPTION ]
```

*list-of-privileges* is one or more of the following privileges:

<table>
<thead>
<tr>
<th>Privilege</th>
<th>Tables</th>
<th>Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INSERT</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DELETE</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>UPDATE [(column-name-list)]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>INDEX</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ALTER</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice that if UPDATE is to be granted, the option (column-name-list) can be used to grant privileges only on certain columns of the specified table or view. If multiple columns are specified, we separate them with commas. If column-name-list is not specified, the privilege(s) is granted for all columns.

*ALL* specifies that we want to grant all privileges that we have on the specified table. *ALL* is *not* applicable to views.

*creator* is the userid of the owner of the table or view on which the privilege(s) are to be granted, followed by a period. It is not necessary for our own tables or views. To grant privileges on another user’s table, we must have been granted the privilege with the GRANT OPTION.

*table-name | view-name* is the name or the table or view on which we want to grant privilege(s).

*list-of-users* is a list of userids, separated by commas, of users to whom we want to grant privileges.
PUBLIC
specifies that we want to grant the privilege(s) specified to all users.

WITH GRANT OPTION
specifies that we want to allow the user(s) specified to be able to grant these privileges to someone else.

Example:
The following command gives the update privilege on the NR_CHILDREN column of the PRES_MARRIAGE table and the select privilege on the entire PRES_MARRIAGE table to user Reagan. It also allows Reagan to pass these privileges on to others.

```
GRANT UPDATE (NR_CHILDREN), SELECT
ON PRES_MARRIAGE
TO REAGAN
WITH GRANT OPTION
```

Format 2 of the GRANT command is used to grant special privileges to other users. In order to use format 2 of the GRANT command, you must have DBA Authority.

The special privileges which may be granted are:

**DBA Authority**
This is the highest level or authorisation provided, and implies CONNECT, RESOURCE and SCHEDULE authorities. A user with DBA authority can perform any operation on any table in the database.

**CONNECT Authority**
CONNECT authority is required to connect to SQL. Granting CONNECT authority in effect creates a new userid.

**RESOURCE Authority**
RESOURCE authority is required to acquire new database files (DBSPACES) and create tables in public DBSPACES.

**SCHEDULE Authority**
SCHEDULE authority allows connection to other userids without specifying a password.

The syntax of the format 2 of the GRANT command is as follows:
Format 2 (For granting special privileges)

```
GRANT { CONNECT TO list-of-users [ IDENTIFIED BY pswrd-list ] | DBA TO list-of-users [ IDENTIFIED BY pswrd-list] | SCHEDULE TO list-of-users [ IDENTIFIED BY pswrd-list ] | RESOURCE TO list-of-users }
```

*list-of-users*
is a list of userids separated by commas. It identifies those users who are to receive the privilege(s).

*pswrds-list*
is a password for each userid listed in list-of-users (separated by commas). For granting DBA or schedule authority, these passwords may be omitted if they have previously been established for all of the users listed.
If DBA Authority is passed to a user, that user also receives connect, resource and schedule authority regardless of whether they are specified on the GRANT command. Furthermore, granting anyone of these privileges to a user without connect authority has the effect that this user automatically receives connect authority.

4.2 REVOKE
If we want to revoke a privilege from a user, we have to use a REVOKE command. A general rule is that a user can only revoke those privileges from another user, which he previously granted to that user. However, any privilege revoked from a user is also revoked from anyone to whom that user may have granted it.
To remove the GRANT OPTION granted by a previous GRANT command, the privilege itself must first be revoked, and then the same privilege without the GRANT OPTION must be re-granted.
The formats of the REVOKE command are very similar to the GRANT command and the parameters have the same meaning:

Format 1  (For revoking privileges on tables and views)
REVOKE { list-of-privileges | ALL }  
ON { creator.} { table-name | view-name }  
FROM { list-of-users | PUBLIC }  

Format 2  (For revoking special privileges)
REVOKE {  
  DBA |  
  CONNECT |  
  RESOURCE |  
  SCHEDULE }  FROM list-of-users

Format 2 of the REVOKE command can be applied only by a user with DBA authority. Using this format, such a user can revoke any special privilege from any user, regardless of who originally granted this special privilege. However, resource or schedule authority cannot be revoked from a user with DBA authority. A user with DBA authority cannot revoke any privileges from himself, therefore ensuring that there is always one user with DBA authority for the database.