This tutorial is designed to introduce you to the skills to implement business rules in a DB2®
database environment. This tutorial will also help you prepare for Section 3 of the DB2 10.1
DBA for Linux, UNIX, and Windows certification exam 611.

Before you start

About this series

Thinking about taking the DB2 DBA certification exam 611? If so, you've come to the right place.
This "DB2 10.1 DBA for Linux, UNIX, and Windows certification exam 611 prep" series is designed
to cover all the topics you need to know before you sit down to take the DB2 10.1 for Linux, UNIX,
and Windows DBA certification exam (exam 611). Even if you're not planning to seek certification
right away, the information presented in this set of tutorials can help you learn about many of the
new features and functionality available in DB2 10.1 for Linux, UNIX, and Windows.

About this tutorial

Eight and one-half percent of the DB2 10.1 for Linux, UNIX, and Windows DBA certification
exam (exam 611) is designed to test your knowledge of how and when to use mechanisms like
constraints, views, and triggers to move the logic used to enforce business rules from application
to a database. The questions that make up this portion of the exam are designed to evaluate the
following:

- Your ability to create constraints on tables.
- Your ability to create views WITH CHECK OPTION.
- Your ability to create and use triggers.
- Your ability to use the SET INTEGRITY command.

This tutorial is designed to introduce the various constraints available with DB2 and show you how
to use these constraints, along with views and triggers, to enforce business rules. This tutorial is
the third in a series of eight tutorials you can use to prepare for exam 611.
Objectives

After completing this tutorial, you should be able to:

- Demonstrate the ability to create constraints on tables.
- Demonstrate the ability to create views with CHECK OPTION.
- Demonstrate the ability to create and use triggers.
- Demonstrate the ability to use the SET INTEGRITY command.

Prerequisites

To take the DB2 10.1 for Linux, UNIX, and Windows DBA certification exam (exam 611), you must have already passed the DB2 10.1 Fundamentals exam (exam 610) or the DB2 9 Fundamentals exam (exam 730).

This tutorial is one tool to help you prepare for exam 611. You should also review the resources at the end of this tutorial for more information about DB2 utilities (see Related topics). Although not all materials discussed in the Fundamentals tutorial series are required to understand the concepts described here, you should at least have a basic knowledge of:

- DB2 products
- DB2 tools
- DB2 instances
- Databases
- Database objects

System requirements

You do not need DB2 to complete this tutorial. However, you will get more out of it if you download the free trial version of IBM DB2 10.1 to work along with this tutorial.

Business rules implementation

Definition

From an information systems perspective, business rules fall into four major categories: definitions of business terms, facts that connect terms to one another, constraints, and derivations. The first two categories (terms and facts) are collectively referred to as structural assertions in the data model. Constraints, on the other hand, are sometimes called action assertions or restrictive rules.

This section focuses on business rules as a statement that defines or constrains some aspect of the business, the implementation of which requires deep knowledge about the business process. For example:

- Every customer must be identified with a valid customer ID.
- Every customer must have a name.
• A customer can have zero, one, or many orders.
• A customer can be associated with only one address.
• A customer must have a valid phone number that is unique.

Such rules must be checked in two distinct system events: when creating a new product/customer or when updating the address of an existing customer.

**Creating constraints on tables**

It is always true that business rules limit the values that can be inserted, deleted, or updated in a table. Though there are several ways to enforce these rules, constraints can guarantee that the business rules are always enforced for all the data due to their direct association with tables.

DB2 10.1 provides the following types of constraints to facilitate the implementation of business rules:

• **NOT NULL**
• **UNIQUE**
• **PRIMARY KEY**
• **FOREIGN KEY**
• **CHECK**
• **Informational**

**NOT NULL constraint**

In a DB2 database, a null value represents an unknown state. By default, null values are allowed for all built in data types. However, there might be some business rules that insist that a value always be provided. For example, "It is mandatory for every customer to have at least one address." Such a rule can be implemented by creating a **NOT NULL** constraint on the address column as shown in the following example.

**Listing 1. NOT NULL constraint example**

```sql
CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT, CUSTOMER_NAME VARCHAR(120), CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500));
```

The **NOT NULL** defined on the CUSTOMER_ADDRESS1 column ensures that it can never be assigned a null value. However, providing values for CUSTOMER_ADDRESS2 is optional as this is a nullable column.

With such a table definition, the following statements will execute successfully:

```sql
INSERT INTO CUSTOMER VALUES (001,'CHRISTINE HAAS','25 EASTCREEK',NULL); INSERT INTO CUSTOMER VALUES (002,'HELENA THOMPSON', '5 ROSEWOOD STREET',' MARKHAM');
```

However, the following statement would fail and return an SQL0407N error, indicating a violation of the **NOT NULL** constraint that has been defined on the CUSTOMER_ADDRESS1 column:
**UNIQUE constraint**

A UNIQUE constraint is a rule that prevents duplicate values from being stored in one or more columns within a table. The columns on which a UNIQUE constraint is needed must be defined as NOT NULL.

UNIQUE constraint can be defined in the `CREATE TABLE` or `ALTER TABLE` statement. The following examples show the creation of a UNIQUE constraint on the CUSTOMER_PHONE column of a table named CUSTOMER.

**Listing 2. UNIQUE constraint with CREATE TABLE example**

```sql
CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_PHONE VARCHAR(12) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT CUST_PH_UNIQ UNIQUE (CUSTOMER_PHONE));
```

**Listing 3. UNIQUE constraint with ALTER TABLE example**

```sql
CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_PHONE VARCHAR(12) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500)); ALTER TABLE CUSTOMER ADD CONSTRAINT CUST_PH_UNIQ UNIQUE (CUSTOMER_PHONE);
```

With such a table definition, the following statement will execute successfully:

```sql
INSERT INTO CUSTOMER VALUES (001,'CHRISTINE HAAS', '493-308-2412',' 25 EASTCREEK',NULL);
```

However, execution of the following statement would result in an SQL0803N error due to the violation of the unique constraint defined on the column CUSTOMER_PHONE.

```sql
INSERT INTO CUSTOMER VALUES (002,'HELENA THOMPSON','493-308-2412', '5 ROSEWOOD STREET',' MARKHAM');
```

Creating a unique constraint automatically creates a unique index on the column. In the event a unique index is created by an ALTER TABLE statement, any existing index on the column will automatically be designated as unique index.

Though uniqueness can be enforced by both a UNIQUE constraint and a unique index, a unique index allows one (and only one) null value on the column(s) specified and generally cannot be used as a parent key to enforce referential integrity. However, a UNIQUE constraint does not allow null values and can be used as a parent key.

**PRIMARY KEY constraint**

A PRIMARY KEY helps identify a row in a table and can be defined on one or more columns of a table. However, there can be only one PRIMARY KEY for a table.

PRIMARY KEYS can be defined when a table is created or can be added later by altering a table's definition.
Listing 4. PRIMARY KEY constraint with CREATE TABLE example

CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_PHONE
VARCHAR(12) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT
P_CUSTOMER PRIMARY KEY (CUSTOMER_ID), CONSTRAINT CUST_PH_UNIQ UNIQUE (CUSTOMER_PHONE));

Listing 5. PRIMARY KEY constraint with ALTER TABLE example

CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_PHONE
VARCHAR(12) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT
CUST_PH_UNIQ UNIQUE (CUSTOMER_PHONE)); ALTER TABLE CUSTOMER ADD CONSTRAINT P_CUSTOMER PRIMARY KEY
(CUSTOMER_ID);

With such a table definition, the following statement will execute successfully (the first time it is executed):

INSERT INTO CUSTOMER VALUES (001,'CHRISTINE HAAS', '493-308-2412',' 25 EASTCREEK',NULL);

However, if the same statement is executed more than once, it results in an SQL0803N error due to a violation of the PRIMARY KEY constraint defined on the CUSTOMER_ID column.

FOREIGN KEY constraint

A business rule such as "Every customer must belong to a branch that is in a list of branches" can be implemented in DB2 Linux, UNIX, and Windows 10.1 by creating a referential constraint between two columns in different tables (for example, the CUSTOMER_BRANCH column of the CUSTOMER table and the BRANCH_ID column of the BRANCH table).

Listing 6. FOREIGN KEY constraint with CREATE TABLE example

CREATE TABLE BRANCH (BRANCH_ID VARCHAR(20) NOT NULL, BRANCH_NAME VARCHAR(50) NOT NULL, BRANCH_MGR
VARCHAR(20) NOT NULL, BRANCH_LOCATION VARCHAR(20) NOT NULL, CONSTRAINT P_BRANCH PRIMARY KEY (BRANCH_ID));
CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_PHONE
VARCHAR(12) NOT NULL, CUSTOMER_BRANCH VARCHAR(20) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL,
CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT P_CUSTOMER PRIMARY KEY (CUSTOMER_ID), CONSTRAINT CUST_PH_UNIQ
UNIQUE (CUSTOMER_PHONE), FOREIGN KEY FK_CUST_BR (CUSTOMER_BRANCH) REFERENCES BRANCH(BRANCH_ID)ON DELETE
RESTRICT);

Listing 7. FOREIGN KEY constraint with ALTER TABLE example

CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_PHONE
VARCHAR(12) NOT NULL, CUSTOMER_BRANCH VARCHAR(20) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL,
CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT P_CUSTOMER PRIMARY KEY (CUSTOMER_ID), CONSTRAINT CUST_PH_UNIQ
UNIQUE (CUSTOMER_PHONE)); ALTER TABLE CUSTOMER ADD CONSTRAINT FK_CUST_BR FOREIGN KEY (CUSTOMER_BRANCH)
REFERENCES BRANCH(BRANCH_ID) ON DELETE RESTRICT;

Here, the BRANCH table acts as the parent table, and the CUSTOMER table acts the child table. The referential constraint defined on the CUSTOMER table allows users/applications to only assign a branch to a customer when the branch exists in the BRANCH table.

Similarly, any branch assigned to a customer should not be deleted from the list of branches. This is enforced by specifying the DELETE rule RESTRICT, which ensures a row in the BRANCH table does not get deleted if referenced by a row in the CUSTOMER table. In addition to RESTRICT, DB2 supports following DELETE rules:
• **NO ACTION**— Same as RESTRICT with difference in when it is enforced.
• **CASCADE**— Deletes all dependent rows. In our example, when a row in BRANCH table is deleted, corresponding rows in CUSTOMER table is also deleted.
• **SET NULL**— All referenced rows will be assigned a null value if nulls are allowed for the column. If a column is defined with the **NOT NULL** constraint, the error code SQL0629N is returned.

Thus, a statement that creates a new branch and inserts details about the branch in BRANCH table might look like this:

```
INSERT INTO BRANCH VALUES ('BR01', 'NATUREVALLEY', 'EVA PULASKI', 'TORONTO');
```

And, a statement that creates a customer and associates the customer with existing branch might look like this:

```
INSERT INTO CUSTOMER VALUES (001,'CHRISTINE HAAS','493-308-2412','BR01','25 EASTCREEK', NULL);
```

This insert operation will be successful because the value specified for the BRANCH column exists in the BRANCH table. However, the following statement results in a SQL0530N error since a corresponding entry is not present in the BRANCH table.

```
INSERT INTO CUSTOMER VALUES (002,'HELENA THOMPSON','413-308-2412', 'BR02', '5 ROSEWOOD STREET',' MARKHAM');
```

**CHECK** constraint

**CHECK** constraint help implement business rules by enforcing data integrity at the table level. For example, a business rule might dictate that the gender of a customer must be either male or female.

Such a business rule can be implemented using a check constraint on the CUSTOMER_GENDER column of the CUSTOMER table as follows:

### Listing 8. **CHECK** constraint with **CREATE TABLE** example

```
CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_GENDER VARCHAR(1) NOT NULL, CUSTOMER_PHONE VARCHAR(12) NOT NULL, CUSTOMER_BRANCH VARCHAR(20) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT P_CUSTOMER PRIMARY KEY (CUSTOMER_ID) , CONSTRAINT CUST_PH_UNIQ UNIQUE (CUSTOMER_PHONE), CONSTRAINT CUST_GEN_CHK CHECK (CUSTOMER_GENDER in ('M','F')), FOREIGN KEY FK_CUST_BR (CUSTOMER_BRANCH) REFERENCES BRANCH(BRANCH_ID)ON DELETE RESTRICT);
```

### Listing 9. **CHECK** constraint with **ALTER TABLE** example

```
CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_GENDER VARCHAR(1) NOT NULL, CUSTOMER_PHONE VARCHAR(12) NOT NULL, CUSTOMER_BRANCH VARCHAR(20) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT P_CUSTOMER PRIMARY KEY (CUSTOMER_ID) , CONSTRAINT CUST_PH_UNIQ UNIQUE (CUSTOMER_PHONE), FOREIGN KEY FK_CUST_BR (CUSTOMER_BRANCH) REFERENCES BRANCH(BRANCH_ID)ON DELETE RESTRICT); ALTER TABLE CUSTOMER ADD CONSTRAINT CUST_GEN_CHK CHECK (CUSTOMER_GENDER in ('M','F'));
```

Any attempt to assign a value other than **M** or **F** results in an SQL0545N error. Thus, the gender of a customer must always be either male or female. For instance, the first two of the following
statements will be completed successfully, whereas the third fails due to violation of the CHECK constraint.

**Listing 10. Violation of CHECK constraint**

| INSERT INTO CUSTOMER VALUES (001,'CHRISTINE HAAS', 'M','493-308-2412','BR01',' 25 EASTCREEK',NULL); INSERT INTO CUSTOMER VALUES (002,'HELENA HOMPSON', 'F','413-467-8732','BR01', '5 ROSEWOOD STREET', 'MARKHAM'); INSERT INTO CUSTOMER VALUES (003,'KATHY SMITH', 'T','417-717-2182','BR01', '1596 BASELINE','AURORA); |

**Informational constraint**

In situations where an application enforces a constraint (such as the gender of a customer), it is needless to enforce the same constraint at the database level. The high amounts of system activity, especially when loading large quantities of records that have referential integrity constraints defined, becomes overhead. Informational constraints avoid this overhead by telling the database manager what rules the data conforms to without the need to be enforced.

**Listing 11. Informational constraint with CREATE TABLE example**

| CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_GENDER VARCHAR(1) NOT NULL, CUSTOMER_PHONE VARCHAR(12) NOT NULL, CUSTOMER_BRANCH VARCHAR(20) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT P_CUSTOMER PRIMARY KEY (CUSTOMER_ID), CONSTRAINT CUST_PH_UNIQ UNIQUE (CUSTOMER_PHONE), CONSTRAINT CUST_GEN_CHK CHECK (CUSTOMER_GENDER in ('M','F')) NOT ENFORCED ENABLE QUERY OPTIMIZATION, FOREIGN KEY FK_CUST_BR (CUSTOMER_BRANCH) REFERENCES BRANCH(BRANCH_ID)ON DELETE RESTRICT); |

**Listing 12. Informational constraint with ALTER TABLE example**

| CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_GENDER VARCHAR(1) NOT NULL, CUSTOMER_PHONE VARCHAR(12) NOT NULL, CUSTOMER_BRANCH VARCHAR(20) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT P_CUSTOMER PRIMARY KEY (CUSTOMER_ID), CONSTRAINT CUST_PH_UNIQ UNIQUE (CUSTOMER_PHONE), CONSTRAINT CUST_GEN_CHK CHECK (CUSTOMER_GENDER in ('M','F')) NOT ENFORCED ENABLE QUERY OPTIMIZATION, FOREIGN KEY FK_CUST_BR (CUSTOMER_BRANCH) REFERENCES BRANCH(BRANCH_ID)ON DELETE RESTRICT); ALTER TABLE CUSTOMER ADD CONSTRAINT CUST_GEN_CHK CHECK (CUSTOMER_GENDER in ('M','F')) NOT ENFORCED ENABLE QUERY OPTIMIZATION; |

These examples use the **NOT ENFORCED** clause to instruct the database manager not to enforce the constraint when **INSERT** and **UPDATE** operations are performed. The **ENABLE QUERY OPTIMIZATION** clause tells the database manager to use the constraint when optimizing SQL operations.

The behavior of **INSERT/UPDATE** operations would need a deep understanding when **NOT ENFORCED** is used.

Consider the following **INSERT** statement:

**Listing 13. Customer insert**

| INSERT INTO CUSTOMER VALUES (001,'CHRISTINE HAAS', 'M','493-308-2412','BR01',' 25 EASTCREEK',NULL), (002,'HELENA THOMPSON', 'F','413-467-8732','BR01', '5 ROSE STREET', 'MARKHAM'), (003,'KATHY SMITH', 'T','417-717-2182','BR01', '1596 BASELINE','AURORA); |

In this example, the value for CUSTOMER_GENDER is specified as T, which violates the check constraint defined for this column. Yet, the database manager allows the insert to happen since the **NOT ENFORCED** clause was used when the constraint was created.
Consequently, the data that resides in the table will look something like this:

**Listing 14. Data that resides in the table**

```
SELECT CUSTOMER_ID, CUSTOMER_NAME, CUSTOMER_GENDER FROM CUSTOMER;
------------------------------------------------- -------------------
1 CHRISTINE HAAS M 2 HELENA THOMPSON F 3 KATHY SMITH T
```

When the same table is queried for the record that violates the check constraint as shown below, unexpected results will be returned.

**Listing 15. Unexpected table results**

```
SELECT CUSTOMER_ID, CUSTOMER_NAME, CUSTOMER_GENDER FROM CUSTOMER WHERE CUSTOMER_GENDER = 'T';
------------------------------------------------- -------------------
0 record(s) selected.
```

This is because of the `ENABLE QUERY OPTIMIZATION` clause, which instructs the database manager to use the constraints during SQL optimization. If this is not the desired result, the constraint needs to be recreated by specifying the `DISABLE QUERY OPTIMIZATION` clause to define the constraint. For example:

```
ALTER TABLE CUSTOMER ALTER CHECK CUST_GEN_CHK DISABLE QUERY OPTIMIZATION;
```

Now the database manager will no longer use the constraint when retrieving data from the table. Consequently, the results produced by querying the table will look something like this:

**Listing 16. Database manager no longer using constraint when retrieving table data**

```
SELECT CUSTOMER_ID, CUSTOMER_NAME, CUSTOMER_GENDER FROM CUSTOMER;
------------------------------------------------- -------------------
1 CHRISTINE HAAS M 2 HELENA THOMPSON F 3 KATHY SMITH T
```

And the query that did not return any records when query optimization was enabled, returns the row that violates the constraint now:

**Listing 17. Query that did not return any records when query optimization was enabled**

```
SELECT CUSTOMER_ID, CUSTOMER_NAME, CUSTOMER_GENDER FROM CUSTOMER WHERE CUSTOMER_GENDER = 'T';
------------------------------------------------- -------------------
3 KATHY SMITH T
```

Informational constraints can help improve performance and can be used if the data originates from a trusted source that guarantees data integrity. Typically, data warehouses that have data originating from different OLTP systems can leverage the informational constraints that are supported in DB2 10.1 for Linux, UNIX, and Windows.

**Creating views with the WITH CHECK OPTION**

In situations where business rules are to be applied to a specific set of users of data, constraints do not always serve the purpose. Placing this logic in the application can be more expensive and
can also impact performance. However, such restrictions can be imposed by using views that have been defined with the `WITH CHECK OPTION` specified. When used, this option enforces restrictions on DML operations performed against a view, based on the predicates that were specified during the view's creation. Thus, if the following SQL statements are executed:

**Listing 18. Creating views with the `WITH CHECK OPTION`**

```sql
CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_GENDER VARCHAR(1) NOT NULL, CUSTOMER_PHONE VARCHAR(12) NOT NULL, CUSTOMER_BRANCH VARCHAR(20) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT P_CUSTOMER PRIMARY KEY (CUSTOMER_ID));
CREATE VIEW CUSTOMER_BR01 (CUSTOMER_ID, CUSTOMER_NAME, CUSTOMER_GENDER, CUSTOMER_PHONE, CUSTOMER_BRANCH, CUSTOMER_ADDRESS1) AS SELECT CUSTOMER_ID, CUSTOMER_NAME, CUSTOMER_GENDER, CUSTOMER_PHONE, CUSTOMER_BRANCH, CUSTOMER_ADDRESS1 FROM CUSTOMER WHERE CUSTOMER_BRANCH = 'BR01' WITH CHECK OPTION;
```

The resulting view—CUSTOMER_BR01—allows the creation of customer records only for the branch 'BR01' and any attempt to assign other branches will result in an error.

Therefore, the following `INSERT` statement succeeds because it conforms to the restrictions imposed by the view:

```
INSERT INTO CUSTOMER_BR01 VALUES (001,'CHRISTINE HAAS', 'M','493-308-2412','BR01',' 25 EASTCREEK');
```

And if the CUSTOMER_BR01 view is queried immediately after this `INSERT` statement is executed, the results produced should look like this:

**Listing 19. CUSTOMER_BR01 when queried immediately**

```sql
SELECT CUSTOMER_ID, CUSTOMER_NAME,CUSTOMER_GENDER FROM CUSTOMER_BR01;
```

On the other hand, the following statement fails and an `SQL0161N` error is returned since the data values provided do not conform to the restrictions imposed by the view's definition.

```
INSERT INTO CUSTOMER_BR01 VALUES (002,'HELENA THOMPSON','F','413-467-8732','BR02','5 ROSE STREET');
```

It is important to note that the `WITH CHECK OPTION` clause cannot be specified for:

- A read-only view
- A view that references the `NODENUMBER` or `PARTITION` function, a nondeterministic function, or a function with an external action
- A typed view

**Creating and using triggers**

In a data management system, constraints and views with check options can be used to implement business rules that are relatively simple. However, triggers are sometimes more useful when it comes to enforcing complex business rules. For example, triggers are best suited for a
business rule that requires writing audit details to an audit table whenever data is changed in a transaction table.

Triggers are used to defining set of actions to be performed in response to events as specified by one or more business rules. Thus, application logic and business rules can be moved into a database by using triggers. This causes such rules to be automatically enforced by the database management system, even when newer applications come on line at a later point in time.

The four major components are associated with triggers:

- The subject on which the trigger is defined (the base table)
- The event that initiates the trigger (INSERT/UPDATE/DELETE)
- The activation time of the trigger (BEFORE OR AFTER the event occurs)
- The action performed by the trigger (transparent to the application)

Using our example of a business rule that calls for an audit trail, if a BEFORE trigger is used, data is inserted in to the audit table before changes are applied to the base table. On the other hand, an AFTER trigger will allow changes to be made to the base table first, then makes entry to the audit table after all changes have been completed. Figure 1 illustrates the behavior of a BEFORE trigger; Figure 2 illustrates the behavior of an AFTER trigger.

Figure 1. How a BEFORE trigger works

Figure 2. How an AFTER trigger works

Suppose you want to create a trigger to enforce a business rule that states, "For any new customer record that gets created, details of the employee that created the customer record, along with the time of record creation, needs to be logged." The first component needed for the trigger is the subject table. For this example, let's suppose it is a table named CUSTOMER that was created as follows.

Listing 20. Creating a table name CUSTOMER

```sql
CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_GENDER VARCHAR(1) NOT NULL, CUSTOMER_PHONE VARCHAR(12) NOT NULL, CUSTOMER_BRANCH VARCHAR(20) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT P_CUSTOMER PRIMARY KEY (CUSTOMER_ID));
```
Based on our business rule, we're also going to need a target table. For this example, let's suppose it is a table named AUDIT_LOG created as follows.

**Listing 21. AUDIT_LOG target table**

```sql
CREATE TABLE AUDIT_LOG (USER_ID VARCHAR(20) NOT NULL, CREATED_CUSTOMER_ID BIGINT NOT NULL, CREATED_TIME TIMESTAMP NOT NULL);
```

The second component needed for the trigger is an event, and in this case, the event is the execution of an **INSERT** statement. The third component needed is the triggering time, which in this example is **AFTER** the **INSERT** operation has been processed. And the fourth component needed is the action to be performed, which in this case is to **INSERT** a record into the AUDIT_LOG table. (The **INSERT** operation happens automatically in response to the triggering event and is transparent to the application.)

Using these four components, we can create the desired trigger:

**Listing 22. Desired trigger**

```sql
CREATE TRIGGER INS_CUST_TRIG AFTER INSERT ON CUSTOMER REFERENCING NEW AS N FOR EACH ROW BEGIN INSERT INTO AUDIT_LOG VALUES (CURRENT USER, N.CUSTOMER_ID,CURRENT TIMESTAMP); END;
```

Once this trigger is created, if the following statement is used to insert new records into the CUSTOMER table (to create new users).

**Listing 23. Insert new records into CUSTOMER table**

```sql
INSERT INTO CUSTOMER VALUES (001,'CHRISTINE HAAS','M','493-308-2412','BR01',' 25 EASTCREEK',NULL),
(002,'HELENA THOMPSON','F','413-467-8732','BR01','5 ROSE STREET',' MARKHAM');
```

Corresponding records are inserted into the AUDIT_LOG table by the trigger. These records can be seen by executing the following query.

**Listing 24. Corresponding records inserted into AUDIT_LOG table by trigger**

```sql
SELECT * FROM AUDIT_LOG; USER_ID CREATED_CUSTOMER_ID CREATED_TIME -------------- ------------------ -------------------------- ADMINISTRATOR 1 2012-09-06-23.02.22.139000 ADMINISTRATOR 2 2012-09-06-23.02.22.139000 2 record(s) selected.
```

Similarly, we can create a trigger that will insert a record into the AUDIT_LOG table trigger any time a record in the CUSTOMER table is modified by executing the following SQL statement.

**Listing 25. Insert record into AUDIT_LOG table trigger when record in CUSTOMER table is modified**

```sql
CREATE TRIGGER UPDT_CUST_TRIG AFTER UPDATE ON CUSTOMER REFERENCING OLD AS O FOR EACH ROW BEGIN INSERT INTO AUDIT_LOG VALUES (CURRENT USER,O.CUSTOMER_ID,CURRENT TIMESTAMP); END;
```

And, once this trigger is created, if the following statement is used to change an existing record into the CUSTOMER table:

```sql
UPDATE CUSTOMER SET CUSTOMER_GENDER = 'F' WHERE CUSTOMER_ID = 001;
```
A corresponding record will be inserted into the AUDIT_LOG table by the trigger. This record can be seen by executing the following query.

**Listing 26. Corresponding record inserted into AUDIT_LOG table by trigger**

```
SELECT * FROM AUDIT_LOG; USER_ID CREATED_CUSTOMER_ID CREATED_TIME
------------------ ----------------- --------------------------
ADMINISTRATOR 1 2012-09-06-23.02.22.139000
ADMINISTRATOR 2 2012-09-06-23.02.22.139000
ADMINISTRATOR 1 2012-09-06-23.03.37.375000
3 record(s) selected.
```

The third record found in the result data set produced was inserted by the **UPDATE** trigger. Finally, we can create a trigger that will insert a record into the AUDIT_LOG table any time a record in the CUSTOMER table is deleted by executing the following SQL statement.

**Listing 27. Create trigger to insert record into AUDIT_LOG table when record in CUSTOMER table is deleted**

```
CREATE TRIGGER DEL_CUST_TRIG AFTER DELETE ON CUSTOMER REFERENCING OLD AS O FOR EACH ROW BEGIN INSERT INTO AUDIT_LOG VALUES (CURRENT USER, O.CUSTOMER_ID, CURRENT_TIMESTAMP); END;
```

Now, if the following statement is used to delete a record from the CUSTOMER table.

**Listing 28. Delete record from CUSTOMER table**

```
DELETE FROM CUSTOMER WHERE CUSTOMER_ID = 001;
```

A corresponding record will be inserted into the AUDIT_LOG table. Once again, this record can be seen by executing the following query.

**Listing 29. Corresponding record inserted into AUDIT_LOG table**

```
SELECT * FROM AUDIT_LOG; USER_ID CREATED_CUSTOMER_ID CREATED_TIME
------------------ ----------------- --------------------------
ADMINISTRATOR 1 2012-09-06-23.02.22.139000
ADMINISTRATOR 2 2012-09-06-23.02.22.139000
ADMINISTRATOR 1 2012-09-06-23.05.14.238000
4 record(s) selected.
```

This time, the fourth record found in the result data set produced was inserted by the **DELETE** trigger.

**Using the SET INTEGRITY command**

Whenever the **LOAD** utility is used on a table, the table is placed in "Check Pending" status. That's because unlike SQL inserts, **LOAD** operations do not perform referential or table constraint checking other than to validate the uniqueness of indices. This is because a **LOAD** operation writes formatted pages directly into the database. The **LOAD** utility uses table states to preserve database consistency; that is, the different states of a table are used to control access to data that has not yet been validated for referential integrity.

When a **LOAD** operation is initiated against a table that has constraints defined, the **LOAD** utility places the table in "Set Integrity Pending" state. This state indicates the table has constraints that were not verified after the last **LOAD**.
The `SET INTEGRITY` transaction control statement can be used to do several things:

- Bring a table out of "Integrity Pending" state by performing the integrity processing required. A table can be incrementally processed by checking only the appended portion for constraints violations. (A table can be checked for constraint violations by executing the `SET INTEGRITY` statement with the `IMMEDIATE_CHECKED` clause specified.)
- Bring a table out of "Integrity Pending" state without performing integrity processing.
- Place a table in "Integrity Pending" state.
- Place a table into full access state.

For example, suppose a relational integrity constraint was created between two tables (BRANCH and CUSTOMER) as follows:

**Listing 30. Creating a relational integrity constraint between two tables: BRANCH and CUSTOMER**

```sql
CREATE TABLE BRANCH (BRANCH_ID VARCHAR(20) NOT NULL, BRANCH_NAME VARCHAR(50) NOT NULL, BRANCH_MGR VARCHAR(20) NOT NULL, BRANCH_LOCATION VARCHAR(20) NOT NULL, CONSTRAINT P_BRANCH PRIMARY KEY (BRANCH_ID));
CREATE TABLE CUSTOMER (CUSTOMER_ID BIGINT NOT NULL, CUSTOMER_NAME VARCHAR(120) NOT NULL, CUSTOMER_GENDER VARCHAR(1) NOT NULL, CUSTOMER_PHONE VARCHAR(12) NOT NULL, CUSTOMER_BRANCH VARCHAR(20) NOT NULL, CUSTOMER_ADDRESS1 VARCHAR(500) NOT NULL, CUSTOMER_ADDRESS2 VARCHAR(500), CONSTRAINT P_CUSTOMER PRIMARY KEY (CUSTOMER_ID), FOREIGN KEY FK_CUST_BR (CUSTOMER_BRANCH) REFERENCES BRANCH(BRANCH_ID) ON DELETE RESTRICT);
```

Now, assume the following statement was used to add two records to the BRANCH table.

```sql
INSERT INTO BRANCH VALUES ('BR01', 'NATUREVALLEY', 'EVA PULASKI', 'TORONTO'), ('BR02', 'GREAT LAKES', 'JAMES JEFFERSON', 'MARKHAM');
```

If the CUSTOMER table were populated using a `LOAD` operation like `LOAD FROM customer.del OF DEL INSERT INTO CUSTOMER;`, an attempt to query the CUSTOMER table immediately after the `LOAD` completes will result in an `SQL0668N` error due to the fact the table is in "Check Pending" status. For example:

**Listing 31. SQL0668N error due to table in "Check Pending" status**

```sql
SELECT CUSTOMER_ID, CUSTOMER_NAME, CUSTOMER_GENDER FROM CUSTOMER;
```

And by querying the system catalog, you can obtain the current status of the CUSTOMER table. For example:

**Listing 32. Obtain current status of CUSTOMER table**

```sql
SELECT TABNAME, STATUS, ACCESS_MODE, CONST_CHECKED FROM SYSCAT.TABLES WHERE TABNAME = 'CUSTOMER';
```

Here, the `STATUS` column indicates the CUSTOMER table is in "Check Pending" state, and the `ACCESS_MODE` column shows the access restriction state relative to the "Check Pending" state.
The value \( N \), which is seen in the first byte in the value for CONST_CHECKED column indicates that the foreign key constraint has not been verified on the data inserted with the \texttt{LOAD} utility.

If the \texttt{SET INTEGRITY} statement is executed with the \texttt{UNCHECKED} option specified, constraint checking is bypassed and the table will remain in "Check Pending" status. For example:

\begin{verbatim}
SET INTEGRITY FOR CUSTOMER CHECK IMMEDIATE UNCHECKED;
\end{verbatim}

On the other hand, if the \texttt{SET INTEGRITY} statement is executed like \texttt{SET INTEGRITY FOR CUSTOMER CHECK IMMEDIATE CHECKED;}, constraint checking is performed and the table is taken out of "Check Pending" status. And by querying the system catalog again, you can verify that the status of the CUSTOMER table has been returned to "Normal." For example:

**Listing 33. Verify that status of CUSTOMER table was returned to "Normal"

\begin{verbatim}
SELECT TABNAME,STATUS,ACCESS_MODE,CONST_CHECKED FROM SYSCAT.TABLES WHERE TABNAME = 'CUSTOMER';
\end{verbatim}

<table>
<thead>
<tr>
<th>TABNAME</th>
<th>STATUS</th>
<th>ACCESS_MODE</th>
<th>CONST_CHECKED</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER</td>
<td>N</td>
<td>F</td>
<td>YYYYYYYYYYYYYYYYYYYYYYYYYYYYYY</td>
</tr>
</tbody>
</table>

As the STATUS column shows, the table is no longer in "Check Pending" status. In addition, the table has been moved to full access state and the validation of foreign keys has been completed, which is evident from the first byte changed to \( Y \) in the value for the CONST_CHECKED column.

At this point, all operations are permissible on the CUSTOMER table, as it has been moved out of "Check Pending" status. Consequently, an attempt to query the CUSTOMER table should be successful.

**Listing 34. Verify that status of CUSTOMER table was returned to "Normal"

\begin{verbatim}
SELECT CUSTOMER_ID, CUSTOMER_NAME, CUSTOMER_GENDER FROM CUSTOMER;
\end{verbatim}

<table>
<thead>
<tr>
<th>CUSTOMER_ID</th>
<th>CUSTOMER_NAME</th>
<th>CUSTOMER_GENDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CHRISTINE HAAS</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>HELENA THOMPSON</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>KATHY SMITH</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>DAVID BROWN</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>RAMLAL MEHTA</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>WING LEE</td>
<td>M</td>
</tr>
<tr>
<td>7</td>
<td>KIYOSHI YAMAMOTO</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>ROY ALONZO</td>
<td>M</td>
</tr>
<tr>
<td>9</td>
<td>KIM NATZ</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>THEODORE SPENSER</td>
<td>M</td>
</tr>
</tbody>
</table>
Related topics

- Read the Preparation Guide for DB2 10.1 DBA for Linux, UNIX, and Windows Exam 611 to learn in-depth information about each of the concepts presented in this tutorial. This guide is compilation of topics from the DB2 10.1 documentation.
- Use the DB2 documentation in IBM Knowledge Center to find more details about each of the concepts presented in this tutorial.
- Use an RSS feed to request notification for the upcoming tutorials in this series. (Find out more about RSS feeds of developerWorks content.)
- Now you can use DB2 for free. Download DB2 Express-C, a no-charge version of DB2 Express Edition for the community that offers the same core data features as DB2 Express Edition and provides a solid base to build and deploy applications.

© Copyright IBM Corporation 2012
Trademark
(www.ibm.com/developerworks/ibm/trademarks/)