



# **IBM DB2® 9.7**

## **Introduction to SQL and database objects Hands-on Lab**

**Information Management Cloud Computing Center of Competence**

**IBM Canada Lab**

---

## Contents

<b>CONTENTS .....</b>	<b>2</b>
<b>1. INTRODUCTION .....</b>	<b>3</b>
<b>2. OBJECTIVES .....</b>	<b>3</b>
<b>3. SUGGESTED READING.....</b>	<b>3</b>
<b>4. GETTING STARTED.....</b>	<b>4</b>
4.1 ENVIRONMENT SETUP REQUIREMENTS .....	4
4.2 INITIAL STEPS.....	4
<b>5. WORKING WITH DB2 DATA OBJECTS .....</b>	<b>5</b>
5.1 TABLES.....	5
5.1.1 SCHEMAS .....	9
5.2 VIEWS .....	11
5.3 ALIASES.....	13
5.4 INDEXES .....	15
5.5 SEQUENCES.....	17
<b>6. WORKING WITH SQL .....</b>	<b>20</b>
6.1 QUERYING DATA .....	20
6.1.1 RETRIEVING ALL ROWS FROM A TABLE USING DATA STUDIO.....	20
6.1.2 RETRIEVING ROWS USING THE SELECT STATEMENT .....	22
6.1.3 SORTING THE RESULTS.....	25
6.1.4 AGGREGATING INFORMATION.....	26
6.1.5 RETRIEVING DATA FROM MULTIPLE TABLES (JOINS) .....	27
6.2 INSERT, UPDATE AND DELETE.....	31
6.2.1 INSERT .....	31
6.2.2 UPDATE .....	33
6.2.3 DELETE .....	34
<b>7. SUMMARY .....</b>	<b>34</b>
<b>8. SOLUTIONS.....</b>	<b>35</b>

---

## 1. Introduction

This module is designed to provide you with an overview of the various objects that can be developed once a database has been created. It also introduces you to SQL, and allows you to practice with SQL statements in Data Studio.

---

## 2. Objectives

By the end of this lab, you will be able to:

- ▶ Examine and manipulate objects within a database
- ▶ Practice with SQL statements

---

## 3. Suggested reading

### **Getting started with DB2 Express-C eBook (Chapter 8)**

<https://www.ibm.com/developerworks/wikis/display/DB2/FREE+Book+-+Getting+Started+with+DB2+Express-C>

A free eBook that can quickly get you up to speed with DB2

### **Getting started with IBM Data Studio for DB2 (Chapters 1-3)**

<https://www.ibm.com/developerworks/wikis/display/db2oncampus/FREE+ebook+-+Getting+started+with+IBM+Data+Studio+for+DB2>

A free eBook that can quickly get you up to speed with IBM Data Studio

### **Database Fundamentals (Chapter 5)**

<https://www.ibm.com/developerworks/wikis/display/db2oncampus/FREE+ebook+-+Database+fundamentals>

A free eBook that introduces you to the relational model and the SQL language


## 4. Getting Started

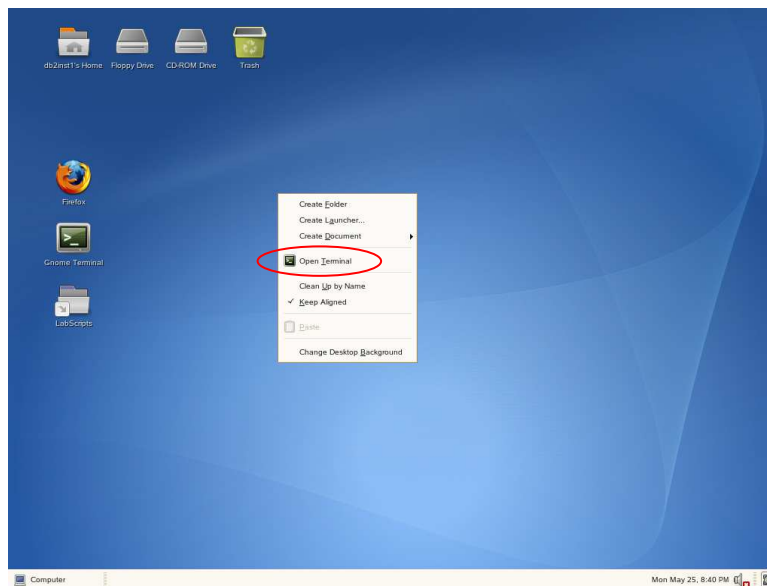
### 4.1 Environment Setup Requirements

To complete this lab you will need the following:

- DB2 Academic Associate Bootcamp VMware image
- VMware Player 2.x or VMware Workstation 5.x or later

### 4.2 Initial Steps

1. Start the VMware image by clicking the  button in VMware.
2. At the login prompt, login with the following credentials:
  - ▶ Username: **db2inst1**
  - ▶ Password: **password**
3. Open a terminal window by right-clicking on the **Desktop** and choosing the **Open Terminal** item.



4. Ensure that the DB2 Database Manager has been started by issuing the following command at the prompt:

```
db2inst1@db2rules:~> db2start
```

**Note:** This command will only work if you logged in as the user `db2inst1`. If you accidentally logged in as another user, type `su - db2inst1` at the command prompt password: `password`.

## 5. Working with DB2 Data Objects

Before we get started with understanding and creating some basic and fundamental database objects, let us create a new database which we will use to highlight some of the concepts within this section.

```
db2inst1@db2rules:~> db2 create db testdb
```

Once the TESTDB database is created, issue a CONNECT statement, as show below, to establish a connection to the newly created database.

```
db2inst1@db2rules:~> db2 connect to testdb
```

### 5.1 Tables

A relational database presents data as a collection of tables. A table consists of data logically arranged in columns and rows (generally known as records).

Tables are created by executing the CREATE TABLE SQL statement. In its simplest form, the syntax for this statement is:

```
CREATE TABLE [TableName]
([ColumnName] [DataType], ...)
```

where:

- `TableName` identifies the name that is to be assigned to the table to be created.
- `ColumnName` identifies the unique name that is to be assigned to the column that is to be created.
- `DataType` identifies the data type to be assigned to the column to be created; the data type specified determines the kind of data values that can be stored in the column.

Thus, if you wanted to create a table named EMPLOYEES that has three columns, one of which is used to store numeric values and two that are used to store character string values, as shown below,

Column	Type
<b>empid</b>	INTEGER
<b>name</b>	CHAR(50)

Dept	CHAR(9)
------	---------

you could do so by executing a CREATE TABLE SQL statement that looks something like this:

```
db2inst1@db2rules:~> db2 "CREATE TABLE employees
      (empid INTEGER,
       name CHAR(50),
       dept INTEGER)"
```

You can execute a DESCRIBE command to view the basic properties of the table:

```
db2inst1@db2rules:~> db2 describe table employees
```

Column name	Data type schema	Data type name	Column Length	Scale	Nulls
EMPID	SYSIBM	INTEGER	4	0	Yes
NAME	SYSIBM	CHARACTER	50	0	Yes
DEPT	SYSIBM	INTEGER	4	0	Yes
3 record(s) selected.					

But, now we notice that the department data type was specified as INTEGER not CHAR as originally intended. Therefore we need a way to change this data type from INTEGER to CHARACTER. We can do this using the alter statement.

### Alter

```
db2inst1@db2rules:~> db2 "alter table employees alter column dept
set data type char(9)"
```

We can view the change by issuing the DESCRIBE command once again:

```
db2inst1@db2rules:~> db2 describe table employees
```

Column name	Data type schema	Data type name	Column Length	Scale	Nulls
EMPID	SYSIBM	INTEGER	4	0	Yes
NAME	SYSIBM	CHARACTER	50	0	Yes
<b>DEPT</b>	<b>SYSIBM</b>	<b>CHARACTER</b>	<b>9</b>	<b>0</b>	<b>Yes</b>
3 record(s) selected.					

Notice now that the DEPT column is now using a CHARACTER data type opposed to an INTEGER data type.

So now that we have our table created to our preference, we can start to input data for the table to hold. We can do some very simple data manipulation language statements, such as insert, update, and delete.

### Insert

Let's insert some basic data into our table with the following statement:

```
db2inst1@db2rules:~> db2 "INSERT INTO employees (EMPID, NAME, DEPT)
VALUES (1, 'Adam', 'A01 '),
(2, 'John', 'B01'),
(3, 'Peter', 'B01'),
(4, 'William', 'A01')"
```

You should receive an SQL0668N message:

```
SQL0668N  Operation not allowed for reason code "7" on table
"DB2INST1.EMPLOYEES".  SQLSTATE=57016
```

What does this mean? If we issue the "? SQL0668N" command, we can view the problem explanation and user response of the particular message.

```
db2inst1@db2rules:~> db2 "? SQL0668N"
```

```
SQL0668N  Operation not allowed for reason code "<reason-code>" on table
"<table-name>".
```

Explanation:

Access to table "<table-name>" is restricted. The cause is based on the following reason codes "<reason-code>":

...

7

The table is in the reorg pending state. This can occur after an ALTER TABLE statement containing a REORG-recommended operation.

...

User response:

...

7

Reorganize the table using the REORG TABLE command.

...

So, now we can conclude that the reason we cannot enter this data is that we did an ALTER on the table previously and it was placed in a reorg pending state. Therefore to resolve this issue, we should do as is recommended and "Reorganize the table using the REORG TABLE command:"

```
db2inst1@db2rules:~> db2 reorg table employees
```

Now try again and issue the insert statement as was shown previously:

```
db2inst1@db2rules:~> db2 "INSERT INTO employees (EMPID, NAME, DEPT)
VALUES (1, 'Adam', 'A01 '),
(2, 'John', 'B01'),
(3, 'Peter', 'B01'),
(4, 'William', 'A01')"
```

To verify the data has been inserted, you can issue a very basic SELECT statement on the table.

```
db2inst1@db2rules:~> db2 "select * from employees"
```

EMPID	NAME	DEPT
-----		
-		
1	Adam	A01
2	John	B01
3	Peter	B01
4	William	A01
4 record(s) selected.		

### Update

We can also make update operations for our table. For example, Peter needs to move from department B01 to department A01. We can make the change in the table with the following update statement:

```
db2inst1@db2rules:~> db2 "update employees set dept='A01' where
name='Peter' "
```

Again, verify that the update has taken place.

```
db2inst1@db2rules:~> db2 "select * from employees"
```

EMPID	NAME	DEPT
-----		
1	Adam	A01
2	John	B01
3	<b>Peter</b>	<b>A01</b>
4	William	A01
4 record(s) selected.		

### Delete

Finally, let's try one more operation on our table, and that is to delete one of the entries. For example, William is no longer one of our employees; therefore we should delete him from our table. We can do so with the following DELETE statement:



```
db2inst1@db2rules:~> db2 "delete employees where name='William'"
```

Again, verify that the delete has taken place.

```
db2inst1@db2rules:~> db2 "select * from employees"
```

EMPID	NAME	DEPT
1	Adam	A01
2	John	B01
3	Peter	A01
3 record(s) selected.		

### 5.1.1 Schemas

A schema is a collection of named objects. Schemas provide a logical classification of objects in the database. A schema can contain tables, views, nicknames, triggers, functions, packages, and other objects.

Most objects in a database are named using a two-part naming convention. The first (leftmost) part of the name is called the schema name or qualifier, and the second (rightmost) part is called the object name. Syntactically, these two parts are concatenated and delimited with a period:

schema\_name.object\_name

A schema is also an object in the database. A schema can be created in 2 ways mainly:

1. It can be implicitly created when another object is created, provided that the user has IMPLICIT\_SCHEMA database authority.
2. It is explicitly created using the CREATE SCHEMA statement with the current user.

Let's take a look at the schema under which our EMPLOYEES table resides. We can do this by listing the tables within our database after we have established a connection.

```
db2inst1@db2rules:~> db2 list tables
```

The output will show a column specifying the Schema which each specific table belongs to:

Table/View	Schema	Type	Creation time
EMPLOYEES	DB2INST1	T	2010-03-30-16.37.05.046385
1 record(s) selected.			

This is an example where the schema will be *implicitly* created when another object is created. So this means when creating our EMPLOYEES table, a schema name was created implicitly as we did not specify any. By default, DB2 is going to use the ID of the user who created the object as the schema name. This is shown in the output above.

Now, as mentioned, we can also *explicitly* create a schema and then assign objects to it upon creation. Let's take an example. We want to create a new schema named MYSCHEMA and create a new table, STORE under this newly created schema. We also want the authorization of this schema to have the authorization ID of our current user (db2inst1).

First, we need to create the schema using the CREATE SCHEMA command:

```
CREATE SCHEMA <name> AUTHORIZATION <name>
```

In our case,

```
db2inst1@db2rules:~> db2 CREATE SCHEMA myschema AUTHORIZATION db2inst1
```

To list all schemas available in the corresponding database you can issue the following command after a connection to the database is established:

```
db2inst1@db2rules:~> db2 select schemaname from syscat.schemata
```

```
SCHEMANAME
```

```
-----
DB2INST1
MYSCHEMA
NULLID
SQLJ
SYSCAT
SYSFUN
SYSIBM
SYSIBMADM
SYSIBMINTERNAL
SYSIBMTS
SYSPROC
SYSPUBLIC
SYSSTAT
SYSTOOLS
```

```
14 record(s) selected.
```

Next, we have to create the table which will belong to MYSCHEMA opposed to DB2INST1. We can do this using the following statement:

```
db2inst1@db2rules:~> db2 "CREATE TABLE myschema.store
                        (storeid INTEGER,
                         address CHAR(50))"
```

(Note: The table name specified must be unique within the schema the table is to be created in.)

We can now list the tables for this new schema:

```
db2inst1@db2rules:~> db2 list tables for schema myschema
```

Table/View	Schema	Type	Creation time
-----	-----	-----	-----
STORE	MYSHEMA	T	2010-03-31-00.16.27.223473
1 record(s) selected.			

We could have also issued the following command to see tables for ALL schemas:

```
db2inst1@db2rules:~> db2 list tables for all
```

Now, you may be wondering why anyone would want to explicitly create a schema using the CREATE SCHEMA statement. The primary reason for explicitly creating a schema has to do with **access control**. An explicitly created schema has an owner, identified either by the authorization ID of the user who executed the CREATE SCHEMA statement or by the authorization ID provided to identify the owner when the schema was created (db2inst1 in our case). The schema owner has the authority to create, alter, and drop any object stored in the schema; to drop the schema itself; and to grant these privileges to other users.

Finally, besides the benefit of access control, we can also have tables with the same name within a single database. This is because the name of each object needs to be unique only within its schema. Let's take a look.

We already have a table called EMPLOYEES within our db2inst1 schema; now let's create another table named employees but under myschema:

```
db2inst1@db2rules:~> db2 "CREATE TABLE myschema.employees
                          (storeid INTEGER,
                           address CHAR(50) ) "
```

It is successful because it is under a different schema and still within the same database!

In all these examples we used tables, but schemas also apply to objects such as: views, indexes, user-defined data types, user-defined functions, nicknames, packages, triggers, etc.

## 5.2 Views

A *view* is an alternative way of representing data that exists in one or more tables. A view can include all or some of the columns from one or more base tables.

In this section we will create a view that will omit certain data from a table, thereby shielding some table data from end users.

In this example, we want to create a view of the EMPLOYEES which will omit the department employee information and rename the first two columns.

Meaning, this is what we want to achieve:

Column	Type
<b>employee_id</b>	INTEGER
<b>first_name</b>	CHAR(50)
<b>Dept</b>	CHAR(9)

To define the view, we must use the CREATE VIEW statement as follows:

```
db2inst1@db2rules:~> db2 "CREATE VIEW empview (employee_id, first_name)
AS SELECT EMPID, NAME
FROM employees"
```

Verify the view has been created:

```
db2inst1@db2rules:~> db2 list tables
```

Table/View	Schema	Type	Creation time
EMPLOYEES	DB2INST1	T	2010-03-30-16.37.05.046385
<b>EMPVIEW</b>	<b>DB2INST1</b>	<b>V</b>	<b>2010-03-31-21.22.26.130570</b>

2 record(s) selected.

Now, describe the view to ensure it is setup the way we originally intended:

```
db2inst1@db2rules:~> db2 describe table empview
```

Column name	Data type schema	Data type name	Column Length	Scale	Nulls
<b>EMPLOYEE_ID</b>	<b>SYSIBM</b>	<b>INTEGER</b>		<b>4</b>	<b>0 Yes</b>
<b>FIRST_NAME</b>	<b>SYSIBM</b>	<b>CHARACTER</b>		<b>50</b>	<b>0 Yes</b>

2 record(s) selected.

This matches what we have initially planned. Now for a final test, let's issue a SELECT \* statement to retrieve all data from the view:

```
db2inst1@db2rules:~> db2 "select * from empview"
```

EMPLOYEE_ID	FIRST_NAME
1	Adam
2	John
3	Peter

3 record(s) selected.

Notice how the column names have changed appropriately as desired, and we cannot receive any data from the department column as we have not included it with our view. We could have also similarly created views which will combine data from different base tables and also based on other views or on a combination of views and tables. We will leave those out of our examples at this point in time but it is important to know that these options are possible.

Although views look similar to base tables, they do not contain real data. Instead, views refer to data stored in other base tables. Only the view definition itself is actually stored in the database. (In fact, when changes are made to the data presented in a view, the changes are actually made to the data stored in the base table(s) the view references.)

For example, update the view and verify that the underlying table contains the corresponding change.

```
db2inst1@db2rules:~> db2 "update empview
      SET FIRST_NAME='Piotr'
      WHERE employee_id=3"
```

Verify.

```
db2inst1@db2rules:~> db2 "SELECT * FROM employees"
```

EMPID	NAME	DEPT
1	Adam	A01
2	John	B01
3	<b>Piotr</b>	A01

3 record(s) selected.

## 5.3 Aliases

Aliases are alternative names for tables or views. An alias can be referenced the same way the table or view the alias refers to can be referenced.

Aliases are publicly referenced names, so no special authority or privilege is required to use them. However, access to the table or view that an alias refers to still requires appropriate authorization.

Aliases can be created by executing the CREATE ALIAS SQL statement.

Let us try and see how to create an alias (named EMPINFO) for our EMPLOYEES table which we have been working with in this section.

```
db2inst1@db2rules:~> db2 CREATE ALIAS empinfo FOR employees
```

Now we have this empinfo alias that we can use to reference the underlying employees table opposed to directly using the table name.

To view this alias, you can issue a command to list the tables:

```
db2inst1@db2rules:~> db2 list tables
```

Table/View	Schema	Type	Creation time
EMPINFO	DB2INST1	A	2010-07-05-11.05.48.124653
EMPLOYEES	DB2INST1	T	2010-07-05-16.37.05.046385
EMPVIEW	DB2INST1	V	2010-07-05-16.30.40.431174

3 record(s) selected.

Let's try a simple select statement with our newly created alias

```
db2inst1@db2rules:~> db2 "SELECT * FROM empinfo "
```

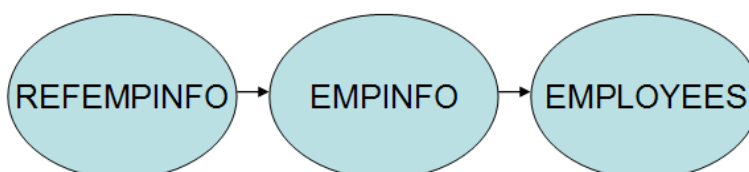
EMPID	NAME	DEPT
1	Adam	A01
2	John	B01
3	Peter	A01

3 record(s) selected.

As you can see, it provides the same output as selecting from the original table name, because after all it is referencing the same table.

Like tables and views, an alias can be created, dropped, and have comments associated with it. *Unlike* tables (but similar to views), aliases can refer to other aliases—a process known as **chaining**. We can take an example of this.

Right now we have our table EMPLOYEES and our EMPINFO alias. We can create this *chain* by creating another alias (REFEMPINFO) which will reference the EMPINFO alias. The situation can be represented by the following diagram:



To do this simply execute the CREATE ALIAS command again, but this time reference the EMPINFO alias opposed to the underlying EMPLOYEES base table name:

```
db2inst1@db2rules:~> db2 CREATE ALIAS refempinfo FOR empinfo
```

List the tables to see this new alias:

```
db2inst1@db2rules:~> db2 list tables
```

Table/View	Schema	Type	Creation time
EMPINFO	DB2INST1	A	2010-07-05-11.05.48.124653
EMPLOYEES	DB2INST1	T	2010-07-05-16.37.05.046385
EMPVIEW	DB2INST1	V	2010-07-05-16.30.40.431174
REFEMPINFO	DB2INST1	A	2010-07-05-16.42.36.059937

4 record(s) selected.

Again we can query this alias which will retrieve the data from the underlying table the name references:

```
db2inst1@db2rules:~> db2 "SELECT * FROM refempinfo"
```

EMPID	NAME	DEPT
1	Adam	A01
2	John	B01
3	Peter	A01

3 record(s) selected.

In conclusion, by using aliases, SQL statements can be constructed in such a way that they are independent of the qualified names that identify the base tables or views they reference.

## 5.4 Indexes

An *index* is an ordered set of pointers to rows of a table. DB2 can use indexes to ensure uniqueness and to improve performance by clustering data, partitioning data, and providing efficient access paths to data for queries. In most cases, access to data is faster with an index than with a scan of the data.

The three main purposes of indexes are:

- To improve performance.
- To ensure that a row is unique.
- To cluster the data.

An index is stored separately from the data in the table. Each index is physically stored in its own index space.

We will work with two examples of indexes in this section. One will illustrate how indexes can benefit to ensure that a row is unique and the other to show how we can improve performance of queries on the database.

In our previous example, we have our EMPLOYEES table with the following structure:

Column	Type
--------	------

<b>empid</b>	INTEGER
<b>name</b>	CHAR(50)
<b>Dept</b>	CHAR(9)

When we created this table, we didn't define any primary key or unique constraints. Thus we can have multiple entries into our table with the same employee ID which is not a situation which we want to have. Therefore, we can use indexes to ensure that there are not two entries in the table with the same value for EMPID:

```
db2inst1@db2rules:~> db2 "CREATE UNIQUE INDEX unique_id
ON employees(empid) "
```

**NOTE:** You will receive the following message if you try to create this unique index with already duplicate entries for the key you are creating the index with. There cannot be duplicate entries when creating a unique index:

```
SQL0603N  A unique index cannot be created because the table
contains data that would result in duplicate index entries.
SQLSTATE=23515
```

Verify the index has been created with the DESCRIBE command:

```
db2inst1@db2rules:~> db2 DESCRIBE INDEXES FOR TABLE employees
```

The output will look similar to the following:

Index schema	Index name	Unique rule	Number of columns	Index type	Index partitioning
DB2INST1	UNIQUE_ID	U	1	RELATIONAL DATA	-
1 record(s) selected.					

The unique rule column determines whether the index is unique or not. There are three different types of unique rules

- D = means duplicate allowed
- P = means primary index
- U = means unique index

Now, let's try to insert a row with a employee ID which already exists (ie, EMPID=3)

```
db2inst1@db2rules:~> db2 "INSERT INTO employees VALUES(3, 'William',
'A01') "
```

You should receive an error message saying that:

```
SQL0803N  One or more values in the INSERT statement, UPDATE
statement, or foreign key update caused by a DELETE statement are
```



```
not valid because the primary key, unique constraint or unique
index identified by "1" constrains table "DB2INST1.EMPLOYEES"
from having duplicate values for the index key.
SQLSTATE=23505
```

This means that our unique index is working properly because we cannot insert a row with an employee ID which already exists (the property on which we defined our index).

Also, as mentioned, index can help improve performance. Something that we can do with indexes is to also collect statistics. Collecting index statistics will allow the optimizer to evaluate whether an index should be used to resolve a query.

We can create an index to collect statistics automatically:

```
db2inst1@db2rules:~> db2 "CREATE INDEX idx
                        ON employees(dept) COLLECT STATISTICS "
```

You can view the index and its properties with the DESCRIBE command as before:

```
db2inst1@db2rules:~> db2 DESCRIBE INDEXES FOR TABLE employees
```

Except for changes in performance, users of the table are unaware that an index is in use. DB2 decides whether to use the index to access the table.

Be aware that indexes have both benefits and disadvantages. A greater number of indexes can simultaneously improve the access performance of a particular transaction and require additional processing for inserting, updating, and deleting index keys. After you create an index, DB2 maintains the index, but you can perform necessary maintenance, such as reorganizing it or recovering it, as necessary.

## 5.5 Sequences

A sequence is an object that is used to generate data values automatically.

Sequences have the following characteristics:

- Values generated can be any exact numeric data type that has a scale of zero.
- Consecutive values can differ by any specified increment value.
- Counter values are recoverable (reconstructed from logs when necessary).
- Values generated can be cached to improve performance.

In addition, sequences can generate values in one of three ways:

- By incrementing or decrementing by a specified amount, without bounds
- By incrementing or decrementing by a specified amount to a user-defined limit and stopping
- By incrementing or decrementing by a specified amount to a user-defined limit, and then cycling back to the beginning and starting again

Let's begin right away with creating a sequence named `emp_id` that starts at 4 and increments by 1, does not cycle, and caches 5 values at a time. To do so, we must issue the following statement:

```
db2inst1@db2rules:~> db2 "CREATE SEQUENCE emp_id
      START WITH 4
      INCREMENT BY 1
      NO CYCLE
      CACHE 5"
```

We will use this sequence to insert a new employee into our table without having to explicitly specify an individual employee ID; the sequence will take care of this for us.

To facilitate the use of sequences in SQL operations, two expressions are available: `PREVIOUS VALUE` and `NEXT VALUE`. The `PREVIOUS VALUE` expression returns the most recently generated value for the specified sequence, while the `NEXT VALUE` expression returns the next sequence value.

Create a new employee named Daniel in department B01 using the `NEXT VALUE` of our newly created sequence:

```
db2inst1@db2rules:~> db2 "INSERT INTO EMPLOYEES
      VALUES (NEXT VALUE FOR emp_id, 'Daniel',
      'B01')"
```

Do a select statement of the table to view the results of how our sequence worked.

```
db2inst1@db2rules:~> db2 "SELECT * FROM employees"
```

EMPID	NAME	DEPT
-----	-----	-----
1	Adam	A01
2	John	B01
3	Piotr	A01
4	<b>Daniel</b>	<b>B01</b>
4 record(s) selected.		

We see that our sequence worked properly. Next time we use the `NEXT VALUE` statement we will increment by 1. For example

```
db2inst1@db2rules:~> db2 "INSERT INTO EMPLOYEES
      VALUES (NEXT VALUE FOR emp_id, 'Stan',
      'B01')"
```

```
db2inst1@db2rules:~> db2 "SELECT * FROM employees"
```

EMPID	NAME	DEPT
-----	-----	-----
1	Adam	A01

2 John	B01
3 Piotr	A01
4 Daniel	B01
5 <b>Stan</b>	<b>B01</b>

5 record(s) selected.

**However**, since we are caching 5 values at a time, we have to be careful because this value identifies the number of values of the identity sequence that are to be pre-generated and kept in memory. (Pre-generating and storing values in the cache reduces synchronous I/O to the log when values are generated for the sequence. However, in the event of a system failure, all cached sequence values that have not been used in committed statements are lost; that is, they can never be used.)

Let's take a look. Terminate the connection and reconnect to the database

```
db2inst1@db2rules:~> db2 terminate
```

```
db2inst1@db2rules:~> db2 connect to testdb
```

Now try the same operation we did previously to add an entry using the sequence and then verify with SELECT \*.

```
db2inst1@db2rules:~> db2 "INSERT INTO EMPLOYEES
VALUES (NEXT VALUE FOR emp_id, 'Bill',
'B01')"
```

```
db2inst1@db2rules:~> db2 "SELECT * FROM employees"
```

EMPID	NAME	DEPT
1 Adam		A01
2 John		B01
<b>9 Bill</b>		<b>B01</b>
3 Piotr		A01
4 Daniel		B01
5 Stan		B01

6 record(s) selected.

Notice that the next value is **9 NOT 6**. Why? Because we specified to cache the next 5 values in the memory before terminating the connection. that is, we had values: 4, 5, 6, 7, 8 in memory. When the connection was lost (system failure) we lost these 5 values and now cannot use them again, thus we had to start the sequence with the next value after those which were cached.

---

## 6. Working with SQL

In this section you will practice with SQL statements. You will use IBM Data Studio and the SAMPLE database. To ensure you have a “clean” copy of the SAMPLE database, let’s drop it and recreate it as follows (from the DB2 Command Window or Linux shell):

```
db2 force applications all
db2 drop db sample
db2 drop db mysample
db2samp1
```

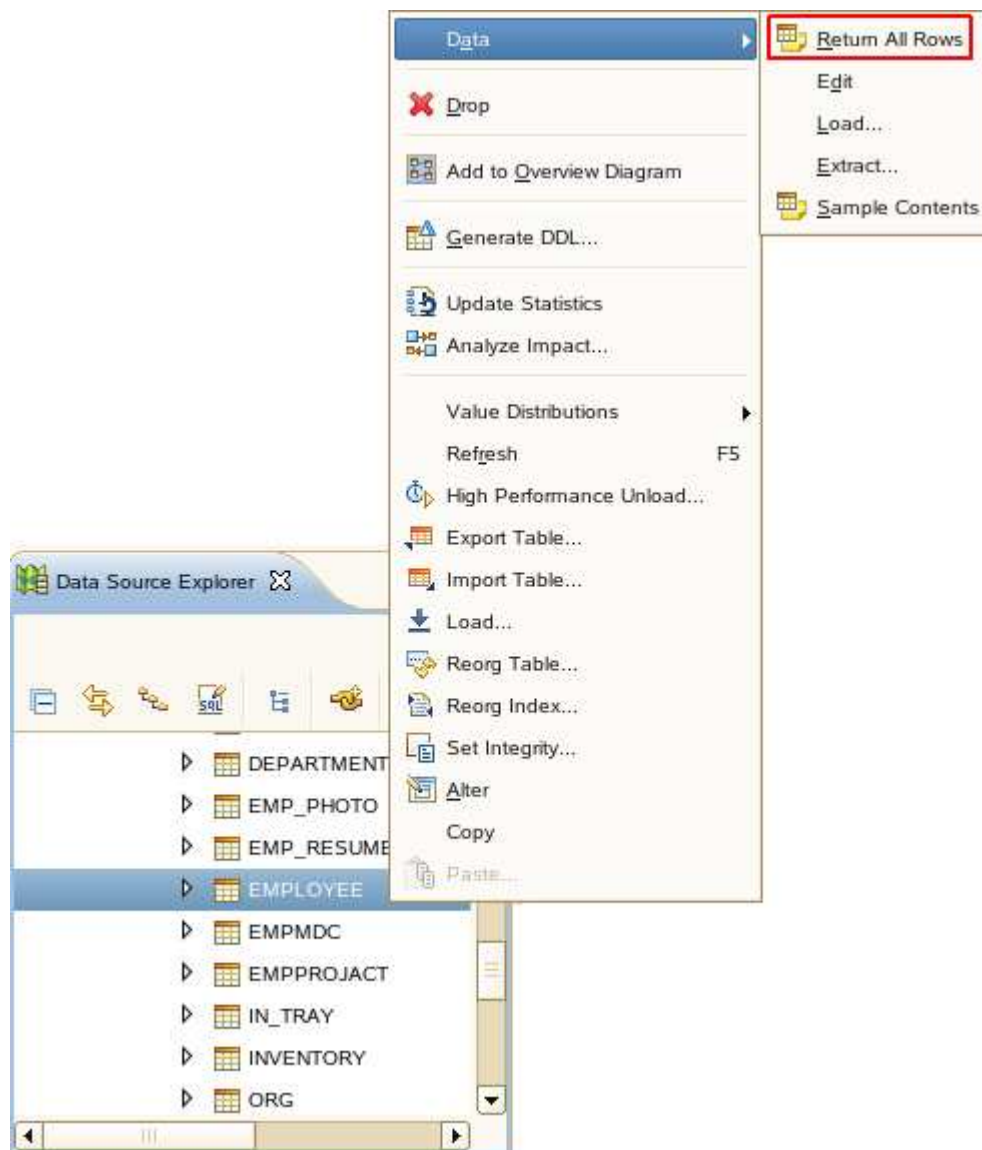
### 6.1 Querying Data

Because no database is worth much unless data can be obtained from it, it’s important to understand how to use a SELECT SQL statement to retrieve data from your tables.

#### 6.1.1 Retrieving all rows from a table using Data Studio

Before using SQL, we’ll quickly show you how to retrieve rows from a table just using the Data Studio options, without the need to write SQL code.

1. In the **Data Source Explorer** view, direct to the table you want to return all the rows. For example, **SAMPLE [DB2 for Linux...] > SAMPLE > Schemas [Filtered] > DB2INST1 > Tables > EMPLOYEE**.
2. Right Click on the table **EMPLOYEE**, choose **Data**, and then click on **Return All Rows**.



3. As we can see under the **SQL Results** tab, the operation was successful. All rows from table **EMPLOYEE** are displayed under the **Result1** tab to the right.

You can always expand or restore views by clicking on the corresponding icons in top right corner of each view.

Status	Operation	Date	Connection Profile
✓ Succeeded	Run SQL	5/14/10 2:23 PM	DATAGURU
✓ Succeeded	SELECT DB2INST1	5/14/10 2:45 PM	DATAGURU
✓ Succeeded	Deploy DB2INST1	5/14/10 2:58 PM	DATAGURU
✓ Succeeded	Run DB2INST1	5/14/10 2:59 PM	DATAGURU
✓ Succeeded	Deploy DB2INST1	5/14/10 3:02 PM	DATAGURU
✓ Succeeded	Run DB2INST1	5/14/10 3:03 PM	DATAGURU
✓ Succeeded	Run SQL	5/14/10 4:00 PM	DATAGURU
✓ Succeeded	"DB2INST1", "S"	5/14/10 4:07 PM	DATAGURU
✓ Succeeded	Sample Content	5/14/10 4:08 PM	DATAGURU
✓ Succeeded	Run SQL	5/14/10 4:14 PM	DATAGURU
✓ Succeeded	Run SQL	5/14/10 4:50 PM	DATAGURU
✓ Succeeded	Run SQL	5/14/10 5:01 PM	DATAGURU
✓ Succeeded	Return All Rows	5/20/10 12:09 P	SAMPLE
✓ Succeeded	Return All Rows	5/20/10 12:10 P	SAMPLE
✓ Succeeded	Sample Content	5/20/10 12:10 P	SAMPLE
✓ Succeeded	Sample Content	5/20/10 12:11 P	SAMPLE
✗ Failed	Run SQL	5/20/10 12:14 P	SAMPLE
✓ Succeeded	Run SQL	5/20/10 12:14 P	SAMPLE
✓ Succeeded	Return All Rows	5/20/10 4:53 PM	SAMPLE
✓ Succeeded	Return All Rows	5/21/10 12:33 A	SAMPLE

EMPNO	FIRSTNAME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE
1	CHRISTINE	I	HAAS	A00	3978	1995-01-01
2	MICHAEL	L	THOMPSON	B01	3476	2003-10-10
3	SALLY	A	KWAN	C01	4738	2005-04-05
4	JOHN	B	GEYER	E01	6789	1979-08-17
5	IRVING	F	STERN	D11	6423	2003-09-14
6	EVA	D	PULASKI	D21	7831	2005-09-30
7	EILEEN	W	HENDERSON	E11	5498	2000-08-15
8	THEODORE	Q	SPENSER	E21	0972	2000-06-19
9	VINCENZO	G	LUCCHESI	A00	3490	1988-05-16
10	SEAN		O'CONNELL	A00	2167	1993-12-05
11	DELORES	M	QUINTANA	C01	4578	2001-07-28
12	HEATHER	A	NICHOLLS	C01	1793	2006-12-15
13	BRUCE		ADAMSON	D11	4510	2002-02-12
14	ELIZABETH	R	PIANKA	D11	3782	2006-10-11
15	MASATOSH	J	YOSHIMURA	D11	2890	1999-09-15
16	MARILYN	S	SCOUTTEN	D11	1682	2003-07-07
17	JAMES	H	WALKER	D11	2986	2004-07-26
18	DAVID		BROWN	D11	4501	2002-03-03
19	WILLIAM	T	JONES	D11	0942	1998-04-11
20	JENNIFER	K	LUTZ	D11	0672	1998-08-29
21	JAMES	J	JEFFERSON	D21	2094	1996-11-21
22	SALVATORI	M	MARINO	D21	3780	2004-12-05
23	DANIEL	S	SMITH	D21	0961	1999-10-30

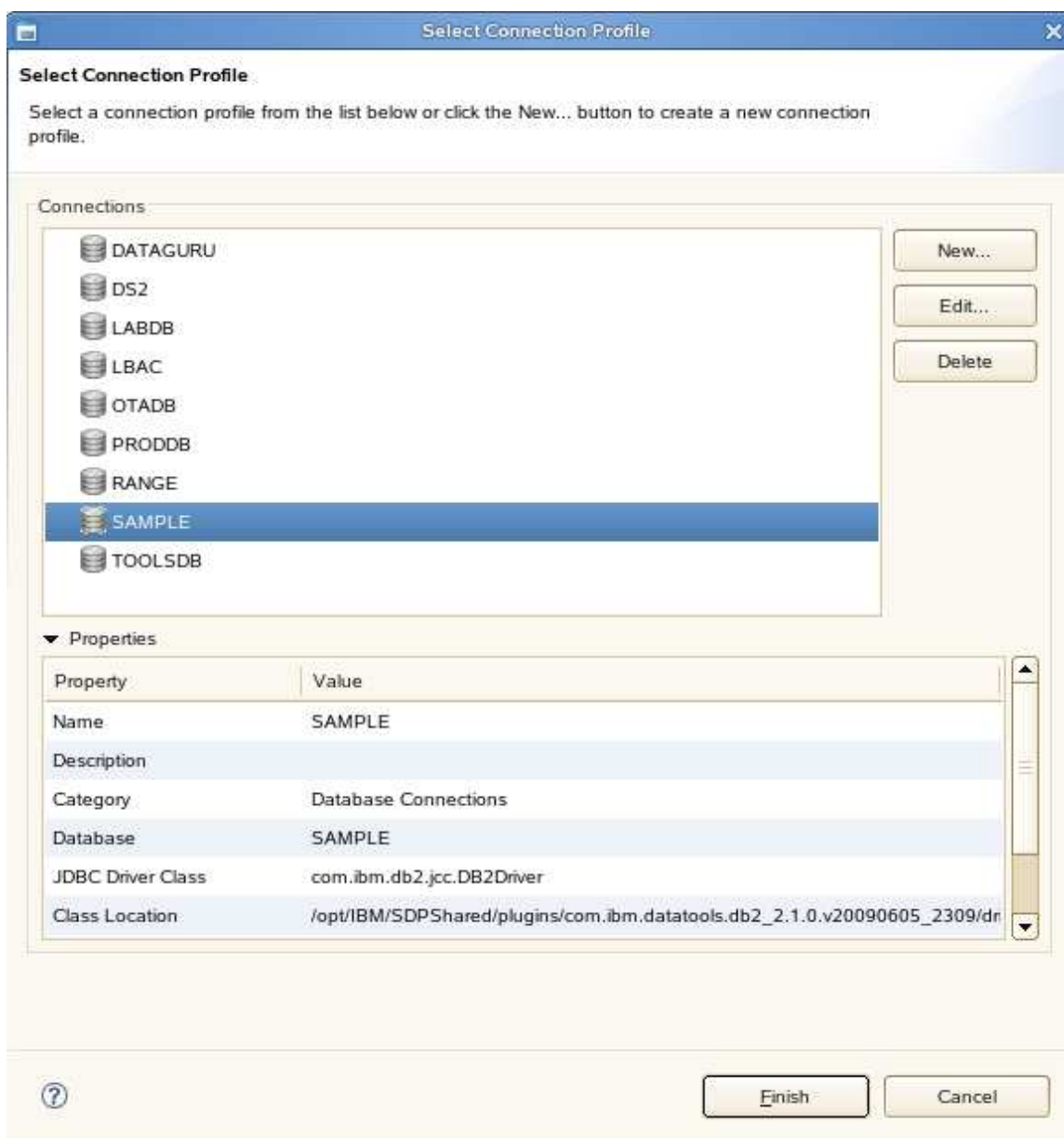
## 6.1.2 Retrieving rows using the SELECT Statement

Follow the steps below to learn how to execute a SELECT statement in Data Studio.

1. In the **Data Source Explorer** toolbar, click the icon **New SQL Script**.



2. In the **Select Connection Profile** window that appears, select **SAMPLE** and click **Finish**.



3. A new tab will appear in the main view. Now let's write a SQL query using a **WHERE** clause, say, we are curious about the BONUS money the managers in department D11 will get. Type in the query below in the newly-created tab:

```
SELECT EMPNO, FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS
FROM EMPLOYEE e
WHERE e. WORKDEPT = 'D11'
```

4. From the main menu, select **Run > Run SQL**

5. Notice that the **SQL Results** view is brought to the foreground at the bottom of the screen. Click the icon to maximize the view. The SQL Results view should indicate that the SQL Script was successful. In the **Status** tab to the right, a summary of the statements in the script file are listed.



6. To view the results of our SQL query statements, click the **Result1** tab on the right.

Status	Result1					
	EMPNO	FIRSTNME	LASTNAME	WORKDEPT	JOB	BONUS
1	000060	IRVING	STERN	D11	MANAGER	500.00
2	000150	BRUCE	ADAMSON	D11	DESIGNER	500.00
3	000160	ELIZABETH	PIANKA	D11	DESIGNER	400.00
4	000170	MASATOSHI	YOSHIMURA	D11	DESIGNER	500.00
5	000180	MARILYN	SCOUTTEN	D11	DESIGNER	500.00
6	000190	JAMES	WALKER	D11	DESIGNER	400.00
7	000200	DAVID	BROWN	D11	DESIGNER	600.00
8	000210	WILLIAM	JONES	D11	DESIGNER	400.00
9	000220	JENNIFER	LUTZ	D11	DESIGNER	600.00
10	200170	KIYOSHI	YAMAMOTO	D11	DESIGNER	500.00
11	200220	REBA	JOHN	D11	DESIGNER	600.00

7. Restore the **SQL Results** view to its original state, and close the **Script.sql** tab in the main view by clicking its **X** icon.

#### 6.1.2.1 Try it: Practice with the **SELECT** statement

Create the SQL statements for the queries described below. You can then compare your answers with the suggested solutions at the end of this lab.

- Find out all sales information from salesman called “*LEE*” in the “Ontario-South” region from the table **SALES**.
- Find out the name of all the departments that have a manager assigned to it from table **DEPARTMENT**.

*Tip:* departments without a manager have NULL in the column **MGRNO**.



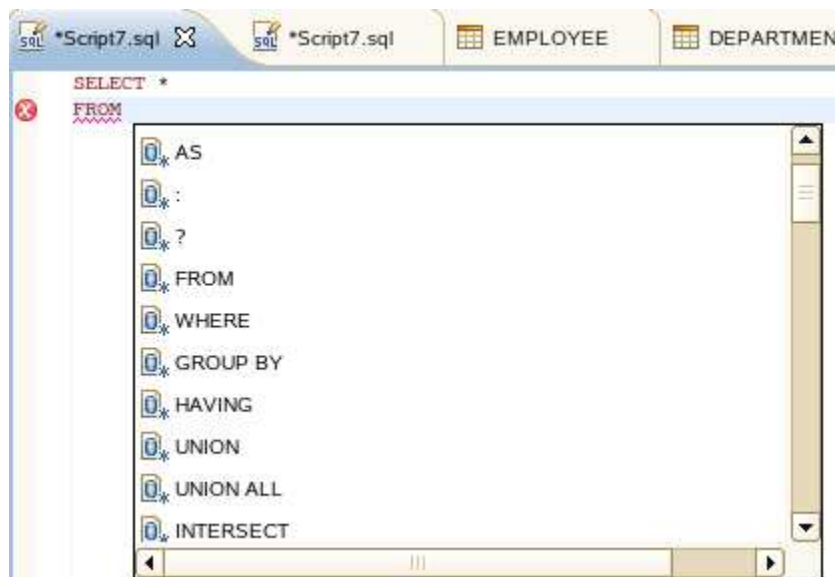
### 6.1.3 Sorting the Results

The **ORDER BY** statement sorts the result set by one or more columns. By default, the records returned by executing a SQL statement are sorted in ascending order, but we can change it to a descending order with the **DESC** keyword.

```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s) ASC|DESC
```

Let's now run an example on our **SAMPLE** database. In the table **STAFF**, rank all the people from department 66 by their salary, in descending order.

**Tip:** You can invoke the code assist function by pressing **Ctrl + Space**. This way instead of typing in the whole words, you can choose from a pre-defined list of keywords.



Now run the query below:

```
SELECT *
FROM STAFF
WHERE DEPT = '66'
ORDER BY SALARY DESC
```

Status	Result1						
	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
1	270	Lea	66	Mgr	9	88555.50	NULL
2	280	Wilson	66	Sales	9	78674.50	811.50
3	320	Gonzales	66	Sales	4	76858.20	844.00
4	310	Graham	66	Sales	13	71000.00	200.30
5	330	Burke	66	Clerk	1	49988.00	55.50

As you can see from the returned table above, manager “Lea” has the highest salary in department 66.

More exercises: (Suggested solutions at the end of the lab)

#### 6.1.3.1 Try it: Practice with the ORDER BY clause

- Using the same table **STAFF** as illustrated above, rank all the people by their years of experience in descending order. For people with same **YEARS**, rank again by their salary in ascending order.

#### 6.1.4 Aggregating Information

The SQL **GROUP BY** statement is used in conjunction with the aggregate functions (e.g. **SUM**, **COUNT**, **MIN**, or **MAX**, etc.) to group the result-set by one or more columns.

```
SELECT column_name(s), aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
```

Consider that you need to find out the average salary for each job position from table **STAFF**. Run the query below:

```
SELECT JOB, AVG(SALARY) as AVG_SALARY
FROM STAFF
GROUP BY JOB
```

The logic behind the SQL is the following: The **GROUP BY JOB** clause instructs DB2 to create groups of rows that have the same value for the **JOB** column, e.g. *Clerk*, *Manager*, etc. Then average salary is calculated by using the function **AVG** applied to the **SALARY** column for each of those group.

- Find the total amount of sales for each sales person in table **SALES**.
- Count the number of male employees for each job position.

SQL joins can be used to query data from two or more tables, based on a relationship between certain columns in these tables.

Before we continue with examples, we will list the types of **JOIN** you can use, and the differences between them.

- 27

4. **FULL (OUTER) JOIN:** Returns all rows that have corresponding PKs and FKs plus the rows from left table that don't have any matches in the right table, plus the rows from right table that don't have any matches in the left table.

Now, from tables **EMP\_PHOTO** and **EMPLOYEE**, let's find out who uploaded a employee photo in bitmap format. Try the query below:

```
SELECT e.EMPNO, p.PHOTO_FORMAT, e.FIRSTNME, e.LASTNAME
FROM EMPLOYEE e, EMP_PHOTO p
WHERE e.EMPNO = p.EMPNO AND p.PHOTO_FORMAT = 'bitmap'
```

**Note:** *p.EMPNO* in **EMP\_PHOTO** is actually **FK** referring to *e.EMPNO*, the **PK** in table **EMPLOYEE**.

Status	Result1			
	EMPNO	PHOTO_FORMAT	FIRSTNME	LASTNAME
1	000130	bitmap	DELORES	QUINTANA
2	000140	bitmap	HEATHER	NICHOLLS
3	000150	bitmap	BRUCE	ADAMSON
4	000190	bitmap	JAMES	WALKER

The rationale behind the SQL above is as follows: First, the tables in the **FROM** clause are combined together into a bigger one. The **WHERE** clause is then responsible for filtering rows from this bigger table. Finally the columns in the **SELECT** clause are returned for all matching rows. This is known as "implicit join notation" for **INNER JOIN**. The equivalent query with "explicit join notation" is shown below:

```
SELECT e.EMPNO, p.PHOTO_FORMAT, e.FIRSTNME, e.LASTNAME
FROM EMPLOYEE e INNER JOIN EMP_PHOTO p
ON e.EMPNO = p.EMPNO
AND p.PHOTO_FORMAT = 'bitmap'
```

Now let's run a similar query, but with **LEFT OUTER JOIN** instead of **INNER JOIN** above. Run the following query:

```
SELECT e.EMPNO, p.PHOTO_FORMAT, e.FIRSTNME, e.LASTNAME
FROM EMPLOYEE e LEFT OUTER JOIN EMP_PHOTO p
ON e.EMPNO = p.EMPNO
AND p.PHOTO_FORMAT = 'bitmap'
```

This time, the result is much longer. From what was explained before, an outer join does not require each record in the two joined tables to have a matching record. And the result of a left (outer) join for table **EMPLOYEE** and **EMP\_PHOTO** always contains all records of the "left" table (**EMPLOYEE**), even if the join-condition does not find any matching record in the "right" table (**EMP\_PHOTO**), and this is where all the *NULL* values under column *PHOTO\_FORMAT* come from.

Status	Result1			
	EMPNO	PHOTO_FORMAT	FIRSTNAME	LASTNAME
1	000130	bitmap	DELORES	QUINTANA
2	000140	bitmap	HEATHER	NICHOLLS
3	000150	bitmap	BRUCE	ADAMSON
4	000190	bitmap	JAMES	WALKER
5	000110	NULL	VINCENZO	LUCCHESSI
6	000180	NULL	MARILYN	SCOUTTEN
7	200010	NULL	DIAN	HEMMINGER
8	200330	NULL	HELENA	WONG
9	000200	NULL	DAVID	BROWN
10	000280	NULL	ETHEL	SCHNEIDER
11	000310	NULL	MAUDE	SETRIGHT
12	300001	NULL	Paul	Pierce
13	000070	NULL	EVA	PULASKI
14	000170	NULL	MASATOSHI	YOSHIMURA
15	000240	NULL	SALVATORE	MARINO
16	000260	NULL	SYBIL	JOHNSON
17	000270	NULL	MARIA	PEREZ
18	200280	NULL	EILEEN	SCHWARTZ
19	000100	NULL	THEODORE	SPENSER
20	000230	NULL	JAMES	JEFFERSON
21	000250	NULL	DANIEL	SMITH
22	000320	NULL	RAMLAL	MEHTA
23	200240	NULL	ROBERT	MONTEVERDE
24	000020	NULL	MICHAEL	THOMPSON
25	000050	NULL	JOHN	GEYER
26	000210	NULL	WILLIAM	JONES
27	000290	NULL	JOHN	PARKER
28	200120	NULL	GREG	ORLANDO
29	200310	NULL	MICHELLE	SPRINGER
30	000010	NULL	CHRISTINE	HAAS
31	000060	NULL	IRVING	STERN
32	000220	NULL	JENNIFER	LUTZ
33	000330	NULL	WING	LEE
34	200220	NULL	REBA	JOHN
35	200340	NULL	ROY	ALONZO
36	000030	NULL	SALLY	KWAN
37	000120	NULL	SEAN	O'CONNELL
38	000300	NULL	PHILIP	SMITH
39	000090	NULL	EILEEN	HENDERSON
40	000160	NULL	ELIZABETH	PIANKA
41	000340	NULL	JASON	GOUNOT
42	200170	NULL	KIYOSHI	YAMAMOTO
43	200140	NULL	KIM	NATZ

### 6.1.5.1 Try it: Practice with JOINS

- Consider you are interested in the action information (with *ACTNO* greater than 100) information and designer names of each project action (from table **PROJECT**). List this information, sorting the results alphabetically according to designers' names.
- Join tables **EMPLOYEE** and **DEPARTMENT**, considering *WORKDEPT* in **EMPLOYEE** is the **FK** referring to *DEPTNO* the **PK** in table **DEPARTMENT**. Save the results, and repeat this query, but use **LEFT OUTER JOIN**, **RIGHT OUTER JOIN** and **FULL OUTER JOIN** instead. Compare the results.

## 6.2 Insert, Update and Delete

Consider now that we are required to enter new product information into our database; or that Ms. Lee gets promoted so her *JOB* and *SALARY* need to be altered accordingly; or Mr. Bryant was not active enough over a certain project and got fired, should we still have him in our employee table?

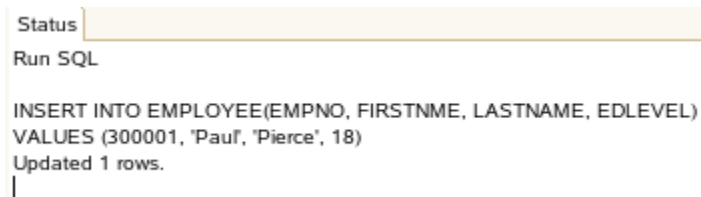
For these situations, we can use the SQL statements **INSERT**, **UPDATE** and **DELETE** to manipulate the table data.

### 6.2.1 INSERT

**INSERT** statement is used to add a new row to a table. For example, NBA player Paul Pierce has retired from his career of basketball player, and successfully locates himself in a position at your company. Now you should add his information to the **EMPLOYEE** table.

Run the query below:

```
INSERT INTO EMPLOYEE(EMPNO, FIRSTNME, LASTNAME, EDLEVEL)
VALUES (300001, 'Paul', 'Pierce', 18)
```



The screenshot shows a SQL execution window. At the top, there is a 'Status' tab. Below it, the text 'Run SQL' is displayed. The SQL statement being executed is:
   
INSERT INTO EMPLOYEE(EMPNO, FIRSTNME, LASTNAME, EDLEVEL)
   
VALUES (300001, 'Paul', 'Pierce', 18)
   
The result of the execution is shown as 'Updated 1 rows.' followed by a vertical bar cursor.

If we run **SELECT \* FROM EMPLOYEE**, we can see that Paul Pierce is now successfully part of the **EMPLOYEE** table, with unspecified columns filled with the default value, which is *NULL* in this case.

Status	Result1													
	EMP	FIRSTNAME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE	JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
1	300001	Paul	NULL	Pierce	NULL	NULL	NULL	NULL 18	NULL	NULL	NULL	NULL	NULL	NULL
2	200340	ROY	R	ALONZO	E21	5698	1997-07-05	FIELT 16	M	1956-05-17	31840.00	500.00	1907.00	
3	200330	HELENA		WONG	E21	2103	2006-02-23	FIELT 14	F	1971-07-18	35370.00	500.00	2030.00	
4	200310	MICHELLE	F	SPRINGER	E11	3332	1994-09-12	OPEF 12	F	1961-04-21	35900.00	300.00	1272.00	
5	200280	EILEEN	R	SCHWARTZ	E11	8997	1997-03-24	OPEF 17	F	1966-03-28	46250.00	500.00	2100.00	
6	200240	ROBERT	M	MONTEVERI	D21	3780	2004-12-05	CLEF 17	M	1984-03-31	37760.00	600.00	2301.00	
7	200220	REBA	K	JOHN	D11	0672	2005-08-29	DESI 18	F	1978-03-19	69840.00	600.00	2387.00	
8	200170	KIYOSHI		YAMAMOTO	D11	2890	2005-09-15	DESI 16	M	1981-01-05	64680.00	500.00	1974.00	
9	200140	KIM	N	NATZ	C01	1793	2006-12-15	ANAL 18	F	1976-01-19	68420.00	600.00	2274.00	
10	200120	GREG		ORLANDO	A00	2167	2002-05-05	CLEF 14	M	1972-10-18	39250.00	600.00	2340.00	
11	200010	DIAN	J	HEMMINGE	A00	3978	1995-01-01	SALE 18	F	1973-08-14	46500.00	1000.00	4220.00	
12	000340	JASON	R	GOUNOT	E21	5698	1977-05-05	FIELT 16	M	1956-05-17	43840.00	500.00	1907.00	

The **INSERT** statement could also have sub-queries, for example, using a **SELECT** clause, which would allow us to insert multiple records at once. Let's try it out. First, execute the DDL below:

```
CREATE TABLE MNG_PEOPLE LIKE EMPLOYEE
```

It creates a new table called **MNG\_PEOPLE** which inherits all the properties/column definitions from table **EMPLOYEE**.

Then we **SELECT** all the managers from table **EMPLOYEE** and insert them into the newly-created table **MNG\_PEOPLE**.

```
INSERT INTO MNG_PEOPLE SELECT * FROM EMPLOYEE WHERE JOB = 'MANAGER'
```

To check if the operation was successful, retrieve all rows from table **MNG\_PEOPLE**. You should see that 7 records were successfully inserted into the new table.

Status	Result1													
	EMPNO	FIRSTNME	MI	LASTNAME	WOR	PHONE	HIRE	JOB	EDL	SEX	BIRTHDATE	SALARY	BONUS	COMM
1	000020	MICHAEL	L	THOMPSON	B01	3476	2003-	MANAGER	18	M	1978-02-02	94250.00	800.00	3300.00
2	000030	SALLY	A	KWAN	C01	4738	2005-	MANAGER	20	F	1971-05-11	98250.00	800.00	3060.00
3	000050	JOHN	B	GEYER	E01	6789	1979-	MANAGER	16	M	1955-09-15	80175.00	800.00	3214.00
4	000060	IRVING	F	STERN	D11	6423	2003-	MANAGER	16	M	1975-07-07	72250.00	500.00	2580.00
5	000070	EVA	D	PULASKI	D21	7831	2005-	MANAGER	16	F	2003-05-26	96170.00	700.00	2893.00
6	000090	EILEEN	W	HENDERSON	E11	5498	2000-	MANAGER	16	F	1971-05-15	89750.00	600.00	2380.00
7	000100	THEODORE	Q	SPENSER	E21	0972	2000-	MANAGER	14	M	1980-12-18	86150.00	500.00	2092.00



### 6.2.1.1 Try it: Practice with INSERT statements

1. Our company just started a new department called *FOO* with department number *K47*, and *'E01'* as ADMRDEPT. Insert this record into table **DEPARTMENT**.
2. Create a new table called **D11\_PROJ** with the same structure of table **PROJECT** and add to it data about all projects from department *D11*.

### 6.2.2 UPDATE

**UPDATE** statement is used to update existing records in a table. Its syntax looks like:

```
UPDATE table_name
SET column1=value, column2=value2,...,column = valueN
WHERE some_column=some_value
```

**Note:** The **WHERE** clause here indicates specifically which record(s) will be updated. Without **WHERE**, all records in this table will be modified!

The Human Resources lady just handed you a detailed information list about Paul Pierce, and asked you to update all the following columns in table **EMPLOYEE** with his data:

```
HIREDATE: 2010-01-01
JOB:      DESIGNER
SEX:      M
BIRTHDATE: 1977-10-13
```

We update his personal information by running the query below:

```
UPDATE EMPLOYEE
SET HIREDATE = '2010-01-01', JOB = 'DESIGNER', SEX = 'M', BIRTHDATE =
'1977-10-13'
WHERE EMPNO = 300001
```

As you can see in the status tab, one row has been updated.

Status
Run SQL
<pre>UPDATE EMPLOYEE SET HIREDATE = '2010-01-01', JOB = 'DESIGNER', SEX = 'M', BIRTHDATE = '1977-10-13' WHERE EMPNO = 300001 Updated 1 rows.</pre>

Again, if we run `SELECT * FROM EMPLOYEE`, we can see that Paul's data has been updated in those four columns.

Status	Result1													
	EMP	FIRSTNAME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE	JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
1	300001	Paul	NULL	Pierce	NULL	NULL	2010-01-01	DESIGNER	18	M	1977-10-13	NULL	NULL	NULL

### 6.2.2.1 Try it: Practice with UPDATE statements

Let's see what else we can do with Paul's information:

- Try to update Paul's EDLEVEL to 'NULL', see what happens.
- Try to update Paul's WORKDEPT to 'Z11', see what happens.

## 6.2.3 DELETE

**DELETE** statement deletes rows in a table. Its syntax looks like:

```
DELETE FROM table_name
      WHERE some_column=some_value
```

**IMPORTANT:** Just like the UPDATE statement, if you omit the **WHERE** clause, all records will be deleted.

Now, let's delete Paul Pierce's record from our database, since he changed his mind and headed back to the basketball court. Run the following query:

```
DELETE FROM EMPLOYEE
      WHERE EMPNO = 300001
```

Check the contents of table **EMPLOYEE** (by now, you should know at least two ways to do so). If you successfully executed the DELETE statement, Paul's record should not be in the result list.

### 6.2.3.1 Try it: Practice with DELETE statements

- Try to delete department 'E21' from table **DEPARTMENT**

---

## 7. Summary

This lab introduced you to the objects that make up a DB2 database. You should now have a solid introduction to DB2 objects in terms of reasoning as to why we

use them and how to create them as well. You also learned and practiced with SQL statements.

## 8. Solutions

### Section 6.1.2.1

#### Query 1

- Find out all sales information from salesman called “*LEE*” in the “Ontario-South” region from the table **SALES**.

```
SELECT *
FROM SALES
WHERE SALES_PERSON = 'LEE'
AND REGION = 'Ontario-South'
```

Status	Result1			
	SALES_DATE	SALES_PERSON	REGION	SALES
1	2005-12-31	LEE	Ontario-South	3
2	2006-03-29	LEE	Ontario-South	2
3	2006-03-30	LEE	Ontario-South	7
4	2006-03-31	LEE	Ontario-South	14
5	2006-04-01	LEE	Ontario-South	8

#### Query 2

- Find out the name of all the departments that have a manager assigned to it from table **DEPARTMENT**.

```
SELECT DEPTNAME
FROM DEPARTMENT
WHERE MGRNO is not NULL
```

Status	Result1
DEPTNAME	
1	SPIFFY COMPUTER SERVICE DIV.
2	PLANNING
3	INFORMATION CENTER
4	MANUFACTURING SYSTEMS
5	ADMINISTRATION SYSTEMS
6	SUPPORT SERVICES
7	OPERATIONS
8	SOFTWARE SUPPORT

### Section 6.1.3.1

#### Query 1

- Using the table **STAFF**, rank all the people by their years of experience in descending order. For people with same **YEARS**, rank again by their salary in ascending order.

```
SELECT *  
FROM STAFF  
WHERE YEARS is not NULL  
ORDER BY YEARS DESC, SALARY ASC
```

Status	Result1						
	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
1	310	Graham	66	Sales	13	71000.00	200.30
2	260	Jones	10	Mgr	12	81234.00	NULL
3	50	Hanes	15	Mgr	10	80659.80	NULL
4	290	Quill	84	Mgr	10	89818.00	NULL
5	210	Lu	10	Mgr	10	90010.00	NULL
6	280	Wilson	66	Sales	9	78674.50	811.50
7	270	Lea	66	Mgr	9	88555.50	NULL
8	190	Sneider	20	Clerk	8	34252.75	126.50
9	20	Pernal	20	Sales	8	78171.25	612.45
10	340	Edwards	84	Sales	7	67844.00	1285.00
11	70	Rothman	15	Sales	7	76502.83	1152.00
12	100	Plotz	42	Mgr	7	78352.80	NULL
13	160	Molinare	10	Mgr	7	82959.20	NULL
14	220	Smith	51	Sales	7	87654.50	992.80
15	10	Sanders	20	Mgr	7	98357.50	NULL
16	90	Koonitz	42	Sales	6	38001.75	1386.70
17	130	Yamaguchi	42	Clerk	6	40505.90	75.60
18	250	Wheeler	51	Clerk	6	74460.00	513.30
19	40	O'Brien	38	Sales	6	78006.00	846.55
20	150	Williams	51	Sales	6	79456.50	637.65
21	140	Fraye	51	Mgr	6	91150.00	NULL
22	110	Ngan	15	Clerk	5	42508.20	206.60
23	350	Gafney	84	Clerk	5	43030.50	188.00
24	300	Davis	84	Sales	5	65454.50	806.10
25	30	Marengi	38	Mgr	5	77506.75	NULL
26	240	Daniels	10	Mgr	5	79260.25	NULL
27	170	Kernisch	15	Clerk	4	42258.50	110.10
28	320	Gonzales	66	Sales	4	76858.20	844.00
29	180	Abrahams	38	Clerk	3	37009.75	236.50
30	230	Lundquist	51	Clerk	3	83369.80	189.65
31	330	Burke	66	Clerk	1	49988.00	55.50
Total 31 records shown							

### Section 6.1.4.1

### Query 1

- Find the total amount of sales for each sales person in table **SALES**.

```
SELECT SALES_PERSON, SUM(SALES) AS total_sales
FROM SALES
GROUP BY SALES_PERSON
```

Status	Result1	
	SALES_PERSON	TOTAL_SALES
1	GOUNOT	50
2	LEE	91
3	LUCCHESI	14

### Query 2

- Count the number of male employees for each job position.

```
SELECT JOB, COUNT(*) as TOTAL_NUM
FROM EMPLOYEE
WHERE SEX = 'M'
GROUP BY JOB
```

Status	Result1	
	JOB	TOTAL_NUM
1	CLERK	6
2	DESIGNER	6
3	FIELDREP	4
4	MANAGER	4
5	OPERATOR	2
6	SALESREP	1

## Section 6.1.5.1

### Query 1

- Consider you are interested in the action information (with *ACTNO* greater than 100) information and designer names of each project action (from table

**PROJECT**). List this information, sorting the results alphabetically according to designers' names.

```
SELECT DISTINCT p.PROJNO, FIRSTNME, LASTNAME, p.ACSTDATE, ep.EMSTDATE
FROM EMPLOYEE e, EMPPROJECT ep, PROJECT p
WHERE e.EMPNO = ep.EMPNO
AND ep.PROJNO = p.PROJNO
AND e.JOB = 'DESIGNER'
AND p.ACTNO > 100
ORDER BY FIRSTNME, LASTNAME
```

Status	Result1				
	PROJNO	FIRSTNME	LASTNAME	ACSTDATE	EMSTDATE
1	MA2112	BRUCE	ADAMSON	2002-07-15	2002-01-01
2	MA2112	BRUCE	ADAMSON	2002-07-15	2002-07-15
3	MA2113	ELIZABETH	PIANKA	2002-10-01	2002-07-15
4	MA2112	JAMES	WALKER	2002-07-15	2002-01-01
5	MA2112	JAMES	WALKER	2002-07-15	2002-10-01
6	MA2113	MARILYN	SCOUTTEN	2002-10-01	2002-04-01
7	MA2112	MASATOSHI	YOSHIMURA	2002-07-15	2002-01-01
8	MA2113	MASATOSHI	YOSHIMURA	2002-10-01	2002-01-01
9	MA2112	MASATOSHI	YOSHIMURA	2002-07-15	2002-06-01
10	MA2113	WILLIAM	JONES	2002-10-01	2002-10-01

## Query 2

- Join tables **EMPLOYEE** and **DEPARTMENT**, considering *WORKDEPT* in **EMPLOYEE** is the **FK** referring to *DEPTNO* the **PK** in table **DEPARTMENT**. Save the results, and repeat this query, but use **LEFT OUTER JOIN**, **RIGHT OUTER JOIN** and **FULL OUTER JOIN** instead. Compare the results.

```
SELECT *
FROM EMPLOYEE e
      INNER
      | LEFT (OUTER)
      | RIGHT (OUTER)
      | FULL (OUTER)
JOIN DEPARTMENT d
```

```
ON e.WORKDEPT = d.DEPTNO
```

### Section 6.2.1.1

#### Query 1

1. Our company just started a new department called *FOO* with department number *K47*, and '*E01*' as ADMRDEPT. Insert this record into table **DEPARTMENT**.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
VALUES ('K47', 'FOO', 'E01')
```

Status	Result1
	DEPTNO    ▲    DEPTNAME    MGRNO    ADMRDEPT    LOCATION
1	K47                    FOO                    NULL                    E01                    NULL

#### Query 2

2. Create a new table called **D11\_PROJ** with the same structure of table **PROJECT** and add to it data about all projects from department *D11*.

```
CREATE TABLE D11_PROJ LIKE PROJECT
```

```
INSERT INTO D11_PROJ
SELECT *
FROM PROJECT
WHERE DEPTNO = 'D11'
```

Status	Result1
	PROJNO    PROJNAME                    DEPTNO    RESPEMP    PRSTAFF    PRSTDATE    PRENDATE    MAJPROJ
1	MA2110    W L PROGRAMMING                    D11                    000060    9.00                    2002-01-01    2003-02-01    MA2100
2	MA2111    W L PROGRAM DESIGN                    D11                    000220    2.00                    2002-01-01    1982-12-01    MA2110
3	MA2112    W L ROBOT DESIGN                    D11                    000150    3.00                    2002-01-01    1982-12-01    MA2110
4	MA2113    W L PROD CONT PROGS                    D11                    000160    3.00                    2002-02-15    1982-12-01    MA2110

### Section 6.2.2.1

#### Query 1

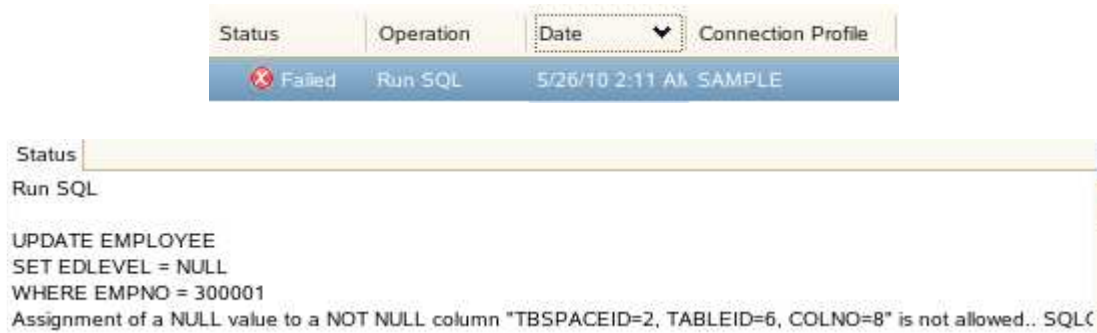
- Try to update Paul's EDLEVEL to '*NULL*', see what happens.

```
UPDATE EMPLOYEE
SET EDLEVEL = NULL
```



```
WHERE EMPNO = 300001
```

Under the SQL Result tab, it says running of this query fails, and the Status tab to the right says it is because we were trying to assign a NULL value to a NOT NULL column, which is illegal.

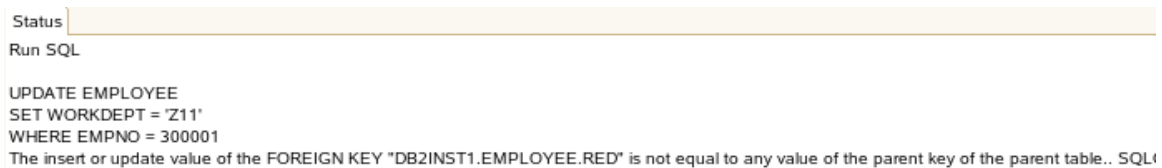


## Query 2

- Try to update Paul's WORKDEPT to 'Z11', see what happens.

```
UPDATE EMPLOYEE
SET WORKDEPT = 'Z11'
WHERE EMPNO = 300001
```

This query failed too, because it tried to update a FOREIGN KEY(FK) column, WORKDEPT in our case, with a value that did not exist in the PRIMARY/PARENT KEY column, column DEPTNO in table DEPARTMENT, that this FK was referring to. This is illegal too.



## Section 6.2.3.1

### Query 1

Try to delete department 'E21' from table **DEPARTMENT**

```
DELETE FROM DEPARTMENT
WHERE DEPTNO = 'E21'
```

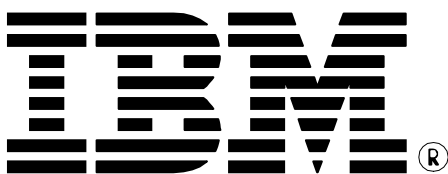
The error message says we can not delete row(s) containing PRIMARY/PARENT KEY column(s) that some other columns are currently referring to, which is, again, illegal.

Status

Run SQL

```
DELETE FROM DEPARTMENT  
WHERE DEPTNO = 'E21'
```

A parent row cannot be deleted because the relationship "DB2INST1.PROJECT.FK\_PROJECT\_1" restricts the deletion..



© Copyright IBM Corporation 2011  
All Rights Reserved.

IBM Canada  
8200 Warden Avenue  
Markham, ON  
L6G 1C7  
Canada

IBM, IBM (logo), and DB2 are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both

UNIX is a registered trademark of The Open Group in the United States, other countries, or both

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates. The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

The information in this publication is provided AS IS without warranty. Such information was obtained from publicly available sources, is current as of April 2010, and is subject to change. Any performance data included in the paper was obtained in the specific operating environment and is provided as an illustration. Performance in other operating environments may vary. More specific information about the capabilities of products described should be obtained from the suppliers of those products.