Chapter 15. Object-Relational Programming

- Triggers
- User-Defined Functions (UDFs)
- Table Functions
- User-Defined Data Types (Distinct Types)
- Object-Relational Extensions
- LOBs and Extenders

This chapter covers some of the more powerful features of DB2’s SQL language. The extended programming features, UDFs, and table functions are discussed along with triggers and user-defined data types (UDTs). The object-relational extensions used to support these features are also covered, and we take a look at how to implement and use large objects (LOBs) and DB2 Extenders.

Triggers

A trigger is a set of actions that will be executed when a defined event occurs. These are known as active triggers. The triggering events can be the following SQL statements:

- INSERT
- UPDATE
- DELETE

Triggers are defined for a specific table and once defined, a trigger is automatically active. A table can have multiple triggers defined for it, and if multiple triggers are defined for a given table, the order of trigger activation is based on the trigger creation timestamp (the order in which the triggers were created). Trigger definitions are stored in the DB2 catalog tables.

- SYSIBM.SYSTRIGGERS
  - One row for each trigger
  - TEXT column contains full text of CREATE TRIGGER
- SYSIBM.SYSPACKAGE
  - One row for each trigger package
  - TYPE column set to T to indicate a trigger package

Trigger Usage

Some of the uses of a trigger include the following:
• Data Validation: Ensures that a new data value is within the proper range. This is similar to table-check constraints, but it is a more flexible data-validation mechanism.

• Data Conditioning: Implemented using triggers that fire before data record modification. This allows the new data value to be modified or conditioned to a predefined value.

• Data Integrity: Can be used to ensure that cross-table dependencies are maintained.

The triggered action could involve updating data records in related tables. This is similar to referential integrity, but it is a more flexible alternative.

We can also use triggers to enforce business rules, create new column values or edit column values, validate all input data, or maintain summary tables or cross-reference tables. They provide for enhanced enterprise and business functionality and faster application development and global enforcement of business rules.

Limited only by our imagination, the trigger is our way of getting control to perform an action whenever a table's data is modified. A single trigger invoked by an update on a financial table could invoke a UDF and/or call a stored procedure to invoke another external action, which triggers an email to a pager to notify the DBA of a serious condition. Farfetched? No, it is already being done.

Triggers can cause other triggers to be invoked and, through the SQL, can call stored procedures. These stored procedures could issue SQL updates that invoke other triggers. This allows great flexibility—we can use triggers to enforce business rules, create new column values or edit column values, validate all input data, or maintain summary tables or cross-reference tables. The trigger is just a way of getting control whenever a table's data is modified.

There is currently a safe limit to the cascading of triggers, stored procedures, and UDFs, which is an execution time nesting depth of 16. This prevents the endless cascading that would be possible. There is a big performance concern here, because if the 17th level is reached, an SQLCODE of -724 is set but all 16 levels are backed out. That could be a significant problem and not something you want to see. The real issue here is processes that are executed outside the control of DB2, since they would not be backed out and it might be very difficult to determine what was changed. There are limitations in the calling sequences; for example, stored procedures that are Workload Manager (WLM)-managed cannot call stored procedures that are DB2-managed.

**Trigger Activation**

A trigger can be defined to fire (be activated) in one of two ways:
• A *before trigger* will fire for each row in the set of affected rows before the triggering SQL statement executes. Therefore, the trigger body is seeing the new data values prior to its being inserted or updated into the table.

• An *after trigger* will fire for each row in the set of affected rows or after the statement has successfully completed (depending on the defined granularity). Therefore, the trigger body is seeing the table as being in a consistent state (i.e., all transactions have been completed).

Another important feature about triggers is that they can fire other triggers (or the same trigger) or other constraints. These are known as *cascading triggers*.

During the execution of a trigger, the new and old data values can be accessible to the trigger, depending on the nature of the trigger (before or after). By using triggers you can:

• Reduce the amount of application development and make development faster. Since triggers are stored in DB2 itself and are processed by DB2, you do not need to code the triggers or their actions into your applications.

• Provide a global environment for your business rules. Since the triggers have to be defined only once and then are stored in the database, they are available to all applications executing against the database.

• Reduce the maintenance of your applications. Since the trigger is handled by DB2 and is stored in the database itself, any changes to the trigger due to changes in your environment have to occur in only one, not multiple, applications.

### Creating Triggers

Triggers are defined using the `CREATE TRIGGER` statement, which contains many options. The primary options are whether it is a before trigger or an after trigger, whether it is a row trigger or a statement trigger, and the language of the trigger. The language is currently only SQL, but that will probably change in the future. There is even rumor that the SQL procedure language is a candidate for triggers. The phrase `MODE DB2SQL` is the execution mode of the trigger. This phrase is required for each trigger to ensure that an existing application will not be negatively impacted if alternative execution modes for triggers are added to DB2 in the future. You can have up to 12 types of triggers on a single table. See Figure 15–1.

**NOTE:** Triggers get invoked in the order they were *created*! A timestamp is recorded when the trigger is created (and recreated). A `DROP` and `(re)CREATE` of a trigger can completely mess up your process by changing the order in which triggers are executed. Be careful!
When adding triggers, the rows that are in violation of a newly added trigger will not be rejected. When a trigger is added to a table that already has existing rows, it will not cause any triggered actions to be activated. If the trigger is designed to enforce some type of integrity constraint on the data rows in the table, those constraints may not be enforced by rules defined in the trigger (or held true) for the rows that existed in the table before the trigger was added.

If an update trigger without an explicit column list is created, packages with an update usage on the target table are invalidated. If an update trigger with a column list is created, packages with update usage on the target table are only invalidated if the package also has an update usage on at least one column in the column-name list of the \texttt{CREATE TRIGGER} statement. If an insert trigger is created, packages that have an insert usage on the target table are invalidated. If a delete trigger is created, packages that have a delete usage on the target table are invalidated.

There is a lot of functionality that can be used within a trigger. For example, a \texttt{CASE} expression can be used in a trigger, but it needs to be nested inside a \texttt{VALUES} statement, as shown here:

\begin{verbatim}
BEGIN ATOMIC
  VALUES CASE
    WHEN condition
      THEN something
    WHEN other condition
      THEN something else
END;
\end{verbatim}
The best method of understanding the usage of triggers is to see some in action. The DB2CERT database contains many relationships that can be maintained using triggers.

### After Trigger

In the following example, a trigger is defined to set the value of the PASS_FAIL column for each of the tests taken by a candidate. (Note that we add this column for this scenario.) The trigger has been given the name PassFail (no relationship with the column called PASS_FAIL). Once the trigger has been created, it is active.

The PassFail trigger is an AFTER, INSERT, and FOR EACH ROW trigger. Every time there is a row inserted into the test_taken table, this trigger will fire. The trigger body section will perform an UPDATE statement to set the value of the PASS_FAIL column for the newly inserted row. The column is populated with either the value P (representing a passing grade) or the value F (representing a failing grade).

**NOTE:** Remember that a trigger defined against one table can modify other tables in the trigger body.

```sql
CREATE TRIGGER PassFail
AFTER INSERT ON db2cert.test_taken
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
UPDATE db2cert.test_taken
SET PASS_FAIL =
CASE
  WHEN n.score >=
  (SELECT cut_score FROM db2cert.test
   WHERE number = n.number)
  THEN 'P'
  WHEN n.score <
  (SELECT cut_score FROM db2cert.test
   WHERE number = n.number)
  THEN 'F'
END
WHERE n.cid = cid
AND n.tcid = tcid
AND n.number = number
AND n.date_taken = date_taken
```
Before Trigger

A before trigger will be activated before the trigger operation has completed. The triggering operation can be on an INSERT, UPDATE, or DELETE statement. This type of trigger is very useful for three purposes:

- To condition data.
- To provide default values.
- To enforce data value constraints dynamically.

There are three before trigger examples shown below that are used in the DB2 Certification application. All three of these triggers have been implemented to avoid seat conflicts for test candidates. The triggers will fire during an insert of each new candidate for a test.

** Example 1 **

```sql
CREATE TRIGGER pre9 NO CASCADE BEFORE INSERT ON db2cert.test_taken
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
WHEN (n.start_time < '09:00:00')
SIGNAL SQLSTATE '70003'
('Cannot assign seat before 09:00:00!')
```

** Example 2 **

```sql
CREATE TRIGGER aft5 NO CASCADE BEFORE INSERT ON db2cert.test_taken
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
WHEN (n.start_time +
(SELECT SMALLINT(length) FROM db2cert.test
WHERE number = n.number) MINUTES
>'17:00:00')
SIGNAL SQLSTATE '70004'
('Cannot assign seat after 17:00:00!')
```

** Example 3 **

```sql
CREATE TRIGGER start NO CASCADE BEFORE INSERT ON db2cert.test_taken
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT cid FROM db2cert.test_taken
WHERE seat_no = n.seat_no
AND tcid = n.tcrid
AND date_taken = n.date_taken
AND n.start_time BETWEEN
    start_time AND finish_time))
SIGNAL SQLSTATE '70001' ('Start Time Conflict!')
```
If the conditions are encountered, an SQL error will be flagged using the SQL function called `SIGNAL`. A different `SQLSTATE` value will be provided when the triggered conditions are encountered.

The `pre9` trigger, shown above, is used to ensure that a test candidate is not scheduled to take a test before 9:00 a.m. The `aft5` trigger is used to ensure that a test candidate is not scheduled to take a test after 5:00 p.m. The `start` trigger is used to avoid conflicts during a testing day.

**Row and Statement Triggers**

In order to understand the concept of trigger granularity, it is necessary to understand the rows affected by the triggering operations. The set of affected rows contains all rows that are deleted, inserted, or updated by the triggering operations.

**Row Triggers**

The keyword `FOR EACH ROW` is used to activate the trigger as many times as the number of rows in the set of affected rows. The previous example shows a row trigger.

**Statement Triggers**

They keyword `FOR EACH STATEMENT` is used to activate the trigger once for the triggering operation.

**Transition Variables and Tables**

We can use transition variables and tables to see before and after images of data effected by trigger execution.

**Transition Variables**

Transition variables allow row triggers to access columns of affected row data in order to see the row data as it existed before the triggering operation and see the row data as it existed after the triggering operation. These variables are implemented by a `REFERENCING` clause in the definition.

```sql
REFERENCING OLD AS OLD_ACCOUNTS
  NEW AS NEW_ACCOUNTS
```

The following example uses transition variables to prevent an update from occurring:

```sql
CREATE TRIGGER TR1 NO CASCADE BEFORE UPDATE ON EMP
```
Here, whenever an update is made to the EMP table, the new value of the salary of the employee, referenced in the before trigger transition variable T1.SALARY is checked against the salary of that employee’s manager. This is done by joining the employee table to the department table, using the transition variable T1.WORKDEPT to get the department information for the employee being updated. Then the EMP table is joined using the manager’s employee number in order to get the salary of the manager.

**Transition Tables**
Transition tables allow after triggers to access a set of affected rows and see how they were before the triggering operation and then see all rows after the triggering operation. Transition tables are also implemented using the REFERENCING clause in the trigger definition.

```
REFERENCING OLD_TABLE AS OLD_ACCT_TABL
NEW_TABLE AS NEW_ACCT_TABLE
```

**NOTE:** Transition tables are materialized in DSNDB07.

Transition tables allow an SQL statement embedded in the trigger body to access the entire set of affected data in the state it was in before or after the change. In the following example, a fullselect reads the entire set of changed rows to pass qualifying data to a user defined function:

```
CREATE TRIGGER EMPMRGR
AFTER UPDATE ON EMP
REFERENCING NEW TABLE AS NTABLE
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
    SELECT SALARYALERT(EMPNO, SALARY)
    FROM NTABLE
    WHERE SALARY > 150000;
END;
```
Transition tables can also be passed to stored procedures and UDFs that are invoked within the body of the trigger. The actual table is not passed as a parameter, but instead a table locator is passed, which can then be used to establish a cursor within the stored procedure or UDF. The following example demonstrates the passing of a transition table to a UDF:

```
CREATE TRIGGER EMPMRGR
AFTER UPDATE ON EMP
REFERENCING NEW TABLE AS NTABLE
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
    VALUES (SALARYALERT(TABLE NTABLE));
END;
```

The corresponding function definition would look something like this:

```
CREATE FUNCTION SALARYALERT (TABLE LIKE EMP AS LOCATOR)
RETURNS INTEGER
EXTERNAL NAME SALERT
PARAMETER STYLE DB2SQL
LANGUAGE C;
```

The C language program would declare a cursor against the transition table by referencing the locator variable that was passed as a parameter in place of a table reference:

```
DECLARE C1 CURSOR FOR
    SELECT EMPNO, SALARY
    FROM TABLE(:LOC1 LIKE EMP)
    WHERE SALARY > 150000;
```

Once the input locator parameter is accepted into the `:LOC1` variable, the cursor can be opened and processed.

### Allowable Combinations

While there are many different combinations of triggers options available, not all are compatible. Table 15–1 shows the valid combinations for trigger options.

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Activation Time</th>
<th>Trigger Operation</th>
<th>Transition Variables Allowed</th>
<th>Transition Tables Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW</td>
<td>BEFORE</td>
<td>INSERT</td>
<td>NEW</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPDATE</td>
<td>OLD, NEW</td>
<td></td>
</tr>
</tbody>
</table>
When a trigger is created, DB2 creates a trigger package. This package is different from packages that you created for an application program (for more information on packages, refer to Chapter 6, “Binding an Application Program”). Trigger packages can be rebound locally, but you cannot bind them (this is done automatically during creation). The package can be rebound only with the `REBIND TRIGGER PACKAGE` command, and this will allow you to change subsets of default bind options (`CURRENTDATA`, `EXPLAIN`, `FLAG`, `ISOLATION`, `RELEASE`). For more information on the bind options, refer to Chapter 6.

Trigger packages cannot be copied, freed, or dropped. In order to delete a trigger package, the `DROP TRIGGER` SQL statement must be issued.

**NOTE:** Rebinding trigger packages after creation is also useful for picking up new access paths.

The qualifier of trigger name determines package collection. For static SQL, the authorization ID of the `QUALIFIER` bind option will be the qualifier, and for dynamic SQL, whatever the `CURRENT SQLID` is will be the qualifier.
Trigger Invalidations

Invalid updates can be detected and stopped by triggers in a couple of ways. You can use the \texttt{SIGNAL SQLSTATE} or the \texttt{RAISE\_ERROR}.

\textbf{SIGNAL SQLSTATE}

\texttt{SIGNAL SQLSTATE} is a new SQL statement that is used to cause an error to be returned to the application with a specified \texttt{SQLSTATE} code and a specific message to stop processing. This statement can be used only as a triggered SQL statement within a trigger and can be controlled with a \texttt{WHEN} clause. The example below shows the use of the \texttt{SIGNAL} statement.

\begin{verbatim}
  WHEN NEW\_ACCT.AMOUNT < (OLD\_ACCT.AMOUNT)
  SIGNAL SQLSTATE '99001' ('Bad amount field')
\end{verbatim}

\textbf{RAISE\_ERROR}

\texttt{RAISE\_ERROR} is not a statement but a built-in function that causes the statement that includes it to return an error with a specific \texttt{SQLSTATE}, \texttt{SQLCODE} \texttt{-438}, and a message. It does basically the same thing as the \texttt{SIGNAL} statement and can be used wherever an expression can be used. The \texttt{RAISE\_ERROR} function always returns null with an undefined data type. \texttt{RAISE\_ERROR} is most useful in \texttt{CASE} expressions, especially when the \texttt{CASE} expression is used in a stored procedure. The following example shows a \texttt{CASE} expression with the \texttt{RAISE\_ERROR} function.

\begin{verbatim}
  VALUES (CASE
    WHEN NEW\_ACCT.AMOUNT < OLD\_ACCT.AMOUNT
    THEN RAISE\_ERROR('99001', 'Bad amount field'))
\end{verbatim}

Forcing a Rollback

If you use the \texttt{SIGNAL} statement to raise an error condition, a rollback will also be performed to back out the changes made by an SQL statement as well as any changes caused by the trigger, such as cascading effects resulting from a referential relationship. \texttt{SIGNAL} can be used in either before or after triggers. Other statements in the program can either be committed or rolled back.

Performing Actions Outside of a Database

Triggers can contain only SQL, but through SQL, stored procedures and UDFs can be invoked. Since stored procedures and UDFs are user-written code, almost any activity can be performed from a triggered event. The action causing the trigger may need a message sent to a special place via email. The trigger might be a before trigger written to handle complex referential integrity checks, which could involve checking if data exists in another non-
DB2 storage container. Through the use of stored procedures and UDFs, the power of a trigger is almost unlimited.

**Performance Issues**

Recursive triggers are updates applied by a trigger causing the same trigger to fire off. These can easily lead to loops and can be very complex statements. However, this may be required by some applications for related rows. You will need code to stop the trigger.

Ordering of multiple triggers can be an issue because triggers on same table are activated in order created (identified in the creation timestamp). The interaction among triggers and referential constraints can also be an issue, because the order of processing can be significant on results produced.

When invoking stored procedures and UDFs from triggers, there are performance and manageability concerns. Triggers can include only SQL but can call stored procedures and UDFs, which are user-written and therefore have many implications on integrity and performance. Transition tables can be passed to stored procedures and UDFs also.

Trigger cascading is when a trigger could modify the triggering table or another table. Triggers can be activated at the same level or different levels, and when activated at different levels, cascading occurs. This can occur only for after triggers. Cascading can occur for UDFs, stored procedures, and triggers. Figure 15–2 shows how to find out how many levels of cascading have occurred. This information can be found in a DB2PM accounting report.

**Monitoring Triggers**

There are various ways to monitor the various actions of triggers. The DB2PM statistics and accounting reports show statistics such as

<table>
<thead>
<tr>
<th>MISCELLANEOUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMNS_BYPASSED 0.0</td>
</tr>
<tr>
<td>MAX_SQL_CASCADE_LEVEL 0.0</td>
</tr>
<tr>
<td>MAX_STORAGE_VALUES 0.0</td>
</tr>
</tbody>
</table>

*Shows the max level Indirect SQL cascading*

*Figure 15–2 Trigger Information in DB2PM Accounting Report*
- The number of times a trigger has been activated.
- The number of times a row trigger was activated.
- The number of times an SQL error occurred during the execution of a triggered action.

Other details can be found in the traces. For example, in IFCID 16 you can find information about the materialization of a work file in support of a transition table where TR is the transition table for triggers. Other information in IFCID 16 includes the depth level of the trigger (0–16), where 0 indicates that there are no triggers. You can also find the type of SQL that invoked the trigger:

I = INSERT,
U = INSERT into a transition table because of an update,
D = INSERT into a transition table because of a delete.

The type of referential integrity (RI) that caused an insert into a transition table for a trigger is also indicated with an S for SET NULL (can occur when the above is a U) or C for CASCADE DELETE (can occur when the above value is D).

If a transition table needs to be scanned for a trigger, you can find this occurrence in IFCID 17: TR for transition table scan for a trigger.

Catalog Information

The SYSIBM.SYSTRIGGERS catalog table contains information about the triggers defined in your databases. To find all the triggers defined on a particular table, the characteristics of each trigger, and to determine the order in which they are executed, you can issue the following query:

```
SELECT DISTINCT SCHEMA, NAME, TRIGTIME, TRIGEVENT,
       GRANULARITY, CREATEDTS
FROM SYSIBM.SYSTRIGGERS
WHERE TBNAME = table-name
  AND TBOWNER = table-owner
ORDER BY CREATEDTS
```

You can get the actual text of the trigger with the following statement:

```
SELECT TEXT, SEQNO
FROM SYSIBM.SYSTRIGGERS
WHERE SCHEMA = schema_name
  AND NAME = trigger_name
ORDER BY SEQNO
```
Triggers Versus Table-Check Constraints

If a trigger and a table-check constraint can enforce the same rule, it is better to use a table-check constraint to enforce business rules. You would want to explore the use of triggers only when a constraint is not enough to enforce a business rule. Constraints and declarative RI are more useful when you have only one state to enforce in a business rule. While triggers are more powerful than table-check constraints and can be more extensive in terms of rule enforcement, constraints can be better optimized by DB2.

Table-check constraints are enforced for all existing data at the time of creation, and are enforced for all statements affecting the data. A table-check constraint is defined on a populated table using the `ALTER TABLE` statement, and the value of the `CURRENT RULES` special register is DB2. Constraints offer a few other advantages over triggers, such as that they are written in a less procedural way than triggers and are better optimized. They protect data against being placed into an invalid state by any kind of statement, whereas a trigger applies only to a specific kind of statement, such as an update or delete.

Triggers are more powerful than check constraints because they can enforce several rules that constraints cannot. You can use triggers to capture rules that involve different states of data, maybe where you need to know the state of the data before and after a calculation.

Triggers and Declarative RI

Trigger operations may result from changes made to enforce DB2 enforced referential constraints. For example, if you are deleting a row from the `EMPLOYEE` table that causes propagated deletes to the `PAYROLL` table through referential constraints, the delete triggers that are defined on the `PAYROLL` table are subsequently executed. The delete triggers are activated as a result of the referential constraint defined on the `EMPLOYEE` table. This may or may not be the desired result, so we need to be aware of cascading effects when using triggers.

Triggers and UDFs

You can use a UDF in a trigger, and these types of functions can help to centralize rules to ensure that they are enforced in the same manner in current and future applications. To invoke a UDF in a trigger, the `VALUES`
Figure 15–3 Invoking a UDF from a Trigger

clause has to be used. Figure 15–3 shows an example of how to invoke a UDF in a trigger.

In the example below, PAGE_DBA is a user-written program, perhaps in C or Java, that formulates a message and triggers a process that sends a message to a pager. By using these kinds of UDFs in triggers, it is possible for a trigger to perform any kind of task and not just be limited to SQL.

BEGIN ATOMIC
VALUES(PAGE_DBA('Table spaces:' CONCAT TS.NAME, 'needs to be reorged NOW!'));
END

UDFs are discussed in more detail later in this chapter.

Object-Relational Extensions

With the object extensions of DB2, you can incorporate object-oriented concepts and methodologies into your relational database by extending DB2 with richer sets of data types and functions. With those extensions, you can store instances of object-oriented data types in columns of tables and operate on them using functions in SQL statements. In addition, you can control the types of operations that users can perform on those data types. The object extensions that DB2 provides are

- Large objects (LOBs)
- Distinct types
- UDFs
Schemas

Schemas are qualifiers used for the object-relational extensions as well as stored procedures. They are used to qualify

- User-Defined Distinct Types
- UDFs
- Stored Procedures
- Triggers

All of the objects qualified by the same schema name can be thought of as a group of related objects. A schema name has a maximum length of 8 bytes. The schema name SYSIBM is used for built-in data types and built-in functions, and SYSPROC is used for some stored procedures delivered by IBM in support of the control center as well as Visual Explain.

Schema Names

The schema name can be specified explicitly when the object is referenced in the CREATE, ALTER, DROP, or COMMENT ON statement. If the object is unqualified and the statement is dynamically prepared, the SQL authorization ID contained in the CURRENT SQLID special register is used for the schema name of the objects.

Schema Privileges

There are certain authorities associated with schemas. Schema privileges include CREATEIN, ALTERIN, and DROPIN. This allows you to create, alter, or drop objects in the identified schema. If the schema name is an authorization ID, then that authorization ID has those privileges implicitly.

PATH Bind Option

The PATH bind option is applicable to BIND PLAN, BIND PACKAGE, REBIND PLAN, and REBIND PACKAGE. The list of schemas specified is used to resolve unqualified references to user-defined distinct types and UDFs in static SQL statements. It is also used to resolve unqualified stored procedure names when the SQL CALL statement specifies a literal for the procedure name. It specifies an ordered list of schemas to be searched to resolve these unqualified references.

CURRENT PATH Special Register

There is also a corresponding special register for the PATH. The SET CURRENT PATH statement changes the value of the PATH special register. This PATH special register is used in the same way as the PATH bind option—to resolve unqualified references in dynamic SQL. It can also be used to resolve unqualified stored procedure names in CALL host-variable statements. The PATH bind option is used to set the initial value for this
special register. SYSIBM and SYSPROC do not need to be specified as part of the PATH—they are implicitly assumed as the first schema.

User-Defined Data Types

We looked briefly at what user-defined data types (UDTs) are and how they are implemented using Data Definition Language (DDL) in Chapter 2, “Database Objects.” In this section we take a closer look at UDTs and some of their benefits.

By using UDTs, you can avoid some excess code in order to support data typing that is not included in the DB2 product. You enable DB2 to do strong-typing, which says that only functions and operations defined on the UDT can be applied to instances of the UDT. This is beneficial for applications so that you do not have to code for comparison errors.

Once the UDT is defined, column definitions can reference that type during the issuing of the CREATE or ALTER statements the same as they would any DB2 built-in data type. If a distinct type is specified without a schema name, the distinct type is resolved by searching the schemas in the current path.

UDTs allow you to use DB2 built-in data types in special ways. UDTs are built off of the DB2 built-in types. UDTs allow you to extend these types and declare specialized usage on them. DB2 then enforces these rules by performing only the kinds of computations and comparisons that you have defined for the data type.

Defining

In order to use a UDT, you must first create it. The UDTs are created by using one of the DB2 built-in types as a base. You create them using the CREATE DISTINCT TYPE statement, as shown below.

```
CREATE DISTINCT TYPE distinct-type-name
   AS source-data-type
   WITH COMPARISONS
```

The name of the distinct type is a two-part name, which must be unique within the DB2 subsystem. The qualifier is a schema name. The distinct type shares a common internal representation with its source data type. However, the distinct type is considered to be an independent data type that is distinct from the others.

An instance of a distinct type can be compared only with another instance of the same distinct type. The WITH COMPARISONS clause is to allow for comparison only between the same distinct type. This phrase is required if the
NOTE: LONG VARCHAR and LONG VARCHAR cannot be used as source types.

source data type is not a LOB type such as BLOB, CLOB, or DBCLOB. Comparisons for these types are not allowed.

NOTE: If you specify the WITH COMPARISONS on a distinct type with an LOB source type, you will get a warning message, but the comparisons are still not allowed.

Casting
There are two operations that are allowed on distinct types: comparisons and casting. You can compare the values of distinct types (non-LOB), or you can cast between the distinct type and the source type.

NOTE: Character and arithmetic operators that are used in built-in functions that are used on a source type are not automatically inherited by the distinct type. These operators and functions need to be created explicitly.

Comparison operators such as those listed in Table 15–2 are allowed on UDTs.

Table 15–2. Comparison Operators for UDTs

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>IN</td>
<td>in</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>between</td>
</tr>
<tr>
<td>NOT IN</td>
<td>not between</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>is not null</td>
</tr>
<tr>
<td>IS NULL</td>
<td>is null</td>
</tr>
</tbody>
</table>

NOTE: LIKE and NOT LIKE are not supported.

Casting functions are used to convert instances of source data types into instances of a different data types. These functions have the name of the
target data type and will have a single parameter, which is the source data type. They will return the target data type. Two cast functions are generated by DB2 when the `CREATE DISTINCT TYPE` is issued. These are used to convert between the distinct type and its source type. They will be created in the same schema as the distinct type. The following shows an example of creating a UDT and then using it both with and without casting:

```
CREATE DISTINCT TYPE EURO AS DECIMAL (9,2) WITH COMPARISONS
EURO (DECIMAL)
   -- where EURO is the target and DECIMAL is the source
DECIMAL (EURO)
   -- where DECIMAL is the target and EURO is the source

Without casting - using the function name
SELECT ITEM
FROM INVENTORY
WHERE COST > EURO (1000.00)

With casting - using cast function
SELECT ITEM
FROM INVENTORY
WHERE COST > CAST (1000.00 AS EURO)
```

**NOTE:** Constants are always considered to be source-type values.

If you want to find all items that have a cost of greater than 1000.00 Euros, you will have to cast, because you cannot compare the data of type `EURO` with data of the source data type of the `EURO`—which is `DECIMAL`. You will need to use the cast function to cast data from `DECIMAL` to `EURO`. You can also use the cast function `DECIMAL` to cast from `EURO` to `DECIMAL` and cast the column `COST` to type `DECIMAL`. Depending on the way you choose to cast—from or to the UDT—you can use the function name notation `data-type(argument)` or the cast notation `CAST(argument AS data-type)`.

**Built-In Functions for UDTs**
The built-in data types come with a collection of built-in functions that operate on them. Some of these functions implement operators, such as the arithmetic operators on numeric data types and substring operators on character data types. Other functions include scalar and column functions, which are discussed in Chapter 4, “Advanced SQL Coding.”

When you create a UDT, you can also specify that it inherit some or all of the functions that operate on the corresponding source type. This is done by creating new functions called `sourced functions` that operate on the UDT and
duplicate the semantics of the built-in functions that work on the source type. The following example shows how to create a sourced function.

```sql
CREATE FUNCTION '+' (EURO, EURO) RETURNS EURO
    SOURCE SYSIBM.'+' (DECIMAL(9,2), DECIMAL(9,2))
```

You can also give UDTs distinct semantics of their own by creating external functions that you write in a host language, which will operate on your UDTs.

**Privileges**

You need to have privileges granted in order to use UDTs. The `GRANT USAGE ON DISTINCT TYPE` is used in order to grant privileges to use the UDT as a column data type in a `CREATE` or `ALTER` statement, or to use the UDT as a parameter in a stored procedure or UDF. The `GRANT EXECUTE ON` enables users to cast functions on a UDT.

**Catalog Information**

There is information about the UDTs stored in the DB2 catalog. The following tables contain information about UDTs.

- `SYSIBM.SYSDATATYPES`: Row for each UDT
- `SYSIBM.SYSROUTINES`: Row for each cast function
- `SYSIBM.SYSROUTINEAUTH`: Authorizations for EXECUTE privilege
- `SYSIBM.SYSRESTAUTH`: Authorizations for USAGE privilege

**UDFs**

UDFs form the basis of object-relational extensions to the SQL language along with UDTs and LOBs. Fundamentally, a database function is a relationship between a set of input data values and a result value. DB2 Universal Database comes with many built-in functions; however, it is possible to create your own column, scalar, and table functions.

While there are three types of functions in DB2, there are only two UDFs: scalar and table. There is no concept of a column function being user-defined.

**NOTE:** The built-in column and scalar functions are covered in Chapter 3, “Basic SQL Coding.”

In DB2, you can create your own scalar or table functions. A UDF can be written in a high-level programming language such as COBOL, C, C++, or Java, or you can use a single SQL statement. There are three types of UDFs:
• Sourced: A UDF based on another scalar UDF or on a built-in scalar or column function. This concept is similar to overloading classes in object-oriented programming.

• SQL Scalar: A scalar UDF that is based upon a single SQL expression. The source code for an SQL UDF is contained entirely within the UDF definition.

• External: A UDF that you write in a supported language. An external UDF can be further categorized as a scalar or table function.

External UDFs
An external UDF is similar to any other program written for the OS/390 or z/OS platform. External UDFs may or may not contain SQL statements, IFI (Instrumentation Facility Interface), or DB2 commands. They may be written in assembler, COBOL, C, C++, PL/I, or Java. External UDFs, once written and generated as dynamically loadable libraries or classes, must be registered with the database. An external function is defined to DB2 with a reference to an OS/390 load module that DB2 should load when the function is invoked. The OS/390 load module contains the object code for the application program that contains the logic of the external function. If the program contains SQL statements, then there is an associated package that contains the DBRM. External functions cannot be column functions. The congrat function shown below is an external scalar UDF and is registered using the CREATE FUNCTION statement.

```sql
CREATE FUNCTION congrat(VARCHAR(30), VARCHAR(40))
RETURNS VARCHAR(30)
EXTERNAL NAME 'CONGRAT'
LANGUAGE C
PARAMETER STYLE DB2SQL
DETERMINISTIC
FENCED
READS SQL DATA
COLLID TEST
NO EXTERNAL ACTION
DISALLOW PARALLEL;
```

DB2 passes parameters to external UDFs in a standard manner much in the same way that parameters are passed to stored procedures. DB2 uses the following structure:

Address of Parameter 1
Address of Parameter 2
Address of Parameter 3

... Address of Result 1
Address of Result 2
Address of Result 3
Address of Parameter 1 Null Indicator
Address of Parameter 2 Null Indicator
Address of Parameter 3 Null Indicator

Address of Result 1 Null Indicator
Address of Result 2 Null Indicator
Address of Result 3 Null Indicator

Address of SQLSTATE
Address of Procedure Name
Address of Specific Name
Address of Message Text
Address of the Scratchpad (if SCRATCHPAD specified in DDL)
Address Call Type Indicator (if FINAL CALL specified in DDL)
Address of DBINFO (if DBINFO specified in DDL)

**NOTE:** A scalar function can return only a single result parameter, while table functions return multiple result parameters each representing a column in a row of the table being returned.

The **SQLSTATE** can be returned from the external UDF to DB2 to indicate a condition that DB2 can then act upon. It is highly recommended that the UDF return a **SQLSTATE** to the caller. The following **SQLSTATEs** can be returned:

- 00000 returns *SQLCODE* 0
- 01Hxx returns *SQLCODE* +462
- 02000 returns *SQLCODE* +100
- 38001 returns *SQLCODE* –487
- 38002 returns *SQLCODE* –577
- 38003 returns *SQLCODE* –751
- 38004 returns *SQLCODE* –579
- 38yxx returns *SQLCODE* –443

If the UDF returns a **SQLSTATE** that is not allowed, DB2 replaces the **SQLSTATE** with 39001, and returns a *SQLCODE* of –463.

**Sourced UDFs**

Sourced UDFs are registered simply by specifying the DB2 built-in source function. Sourced functions can be scalar functions or column functions, but cannot be table functions. Sourced functions are often helpful when there is a need allow for the use of a built-in function on a UDT (see previous section in this chapter for information on UDTs).
This example allows us to create an avg function for the score data type:

```sql
CREATE FUNCTION AVG (SCORE) RETURNS SCORE
    SOURCE SYSIBM.AVG(DECIMAL);
```

These `CREATE FUNCTION` statements place an entry for each UDF in the `SYSIBM.SYSROUTINES` catalog table, and the parameters are recorded in `SYSIBM.SYSPARMS`. These catalog tables can be queried for information about the UDFs.

**SQL Scalar UDFs**

An SQL scalar function is a UDF in which the entire functionality of the function is a single SQL expression and is coded into the `CREATE FUNCTION` statement. The function is identified as an SQL scalar function by coding the `LANGUAGE SQL` option of the `CREATE FUNCTION` statement. This enables you to code an expression used commonly within more than one statement and modularize that expression by storing it separately as a UDF. Any SQL statement can then reference the UDF in the same manner in which any scalar function can be invoked. This enables common expressions to be coded only once and stored separately in the DB2 catalog, centralizing the coding and administration of these types of functions.

The SQL expression is specified in the `RETURN` clause of the `CREATE FUNCTION` statement, and can contain references to the function input parameters as in the following example that computes the total number of months between two dates:

```sql
CREATE FUNCTION TOTMON (STARTX DATE, ENDY DATE) RETURNS INTEGER
    LANGUAGE SQL CONTAINS SQL
    NO EXTERNAL ACTION
    DETERMINISTIC
    RETURN ABS( (YEAR(STARTX - ENDY)*12) + MONTH(STARTX - ENDY) );
```

The expressions contained in the SQL scalar UDF cannot contain references to columns names or host variables. However, an SQL scalar UDF can invoke other UDFs, which may be external UDFs that can be an SQL program.

The source code for an SQL scalar function is actually stored in the `SYSIBM.SYSVIEWS` DB2 catalog table. When an SQL statement referencing a SQL scalar function is compiled, the function source from the `SYSIBM.SYSVIEWS` catalog table is merged into the statement. Package and plan dependencies on the SQL scalar functions, as with all UDFs, are maintained in the `SYSIBM.SYSPACKDEP` and `SYSIBM.SYSPLANDEP` tables respectively.
Table Functions
With DB2, you can also create another type of UDF called a table function. A table function is a UDF that returns a table to the SQL statement that calls it. This means that a table function can only be referenced in the FROM clause of a SELECT statement. The table function provides a means of including external data or complex processes in SQL queries. Table functions can read non-DB2 data—for instance, a file on the operating system or over the World Wide Web—tabularize it, and return the data to DB2 as a relational table that can subsequently be treated like any other relational table. For example, the APPFORM table function in the next example takes in a candidate application form, processes it, and returns the data in an appropriate format to be inserted in the CANDIDATE table (except for the candidate ID, which is generated):

```sql
CREATE FUNCTION APPFORM(VARCHAR(30))
RETURNS TABLE
  (LNAME VARCHAR(30),
   FNAME VARCHAR(30),
   INITIAL CHAR(1),
   HPHONE PHONE,
   WPHONE PHONE,
   STREETNO VARCHAR(8),
   STREETNAME VARCHAR(20),
   CITY VARCHAR(20),
   PROV_STATE VARCHAR(30),
   CODE CHAR(6),
   COUNTRY VARCHAR(20))
EXTERNAL NAME APPFORM
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION
FINAL CALL
DISALLOW PARALLEL
CARDINALITY 20;
```

If we wanted to insert a new candidate into the CANDIDATE table based on his or her application form, we could use the following SELECT statement:

```sql
INSERT INTO CANDIDATE
SELECT CID,
   LNAME, FNAME, INITIAL,
   HPHONE, WPHONE,
   STREETNO, STREETNAME, CITY,
   PROV_STATE, CODE, COUNTRY
FROM TABLE(APPFORM('\DOCS\NEWFORM.TXT')) AS AP
```
Invoking User-Defined Functions

Scalar UDFs are invoked much in the same way as any built-in DB2 scalar function. A function name identifies the function, and one or more parameters pass information from the invoking SQL statement to the UDF. The parameters passed can be table columns, constants, or expressions. If an expression is passed to an external UDF, DB2 resolves the expression and then passes the result to the UDF. The result of the UDF execution replaces the function invocation at execution time. In the following example, the SQL scalar function `TOTMON` is used to calculate the number of months between two dates:

```sql
SELECT HIREDATE, BIRTHDATE,
       TOTMON(HIREDATE, BIRTHDATE) as total_months,
FROM   DSN8710.EMP;
```

Here, the `HIREDATE` and `BIRTHDATE` columns are selected from the `EMP` table, and the `TOTMON` function (previously defined in this chapter) determines the total number of months between the two dates, which were fed to the function as parameters. In this case, the `TOTMON` function, being an SQL scalar function, is merged with the statement during statement compilation as if the expression itself were coded within the SQL statement.

A UDF can be defined as deterministic or not deterministic. A deterministic function is one that will return the same result from one invocation to the next if the input parameter values have not changed. While there is no mechanism within DB2 to “store” the results of a deterministic function, the designation can impact the invoking query execution path relative to materialization. In a situation in which a table expression has been nested within an SQL statement, a nondeterministic function may force the materialization of the inner query. For example,

```sql
SELECT WORKDEPT, SUM(TOTAL_MONTHS), AVG(TOTAL_MONTHS)
FROM (SELECT WORKDEPT,
       TOTMON(HIREDATE, BIRTHDATE) as total_months,
FROM   DSN8710.EMP) AS TAB1
GROUP BY WORKDEPT;
```

Here, if there is an index on the `WORKDEPT` column of the `EMP` table, then the inner table expression called `TAB1` may not be materialized, but rather be merged with the outer select statement. This is possible because the `TOTMON` function is deterministic. If the `TOTMON` function was not deterministic, then DB2 would have to materialize the `TAB1` table expression, possibly storing it in the DSNDB07 temporary table spaces and sorting to perform the desired aggregation. However, its not exactly clear as to whether or not having `TOTMON` be deterministic is a good thing. If the `TOTMON` function is CPU-intensive, it may be better to materialize the result of the inner table.
expression. This is because the merged $TOTMON$ function (if it is deterministic) will actually be executed twice in the outer query, once per reference (for the $SUM$ and $AVG$ functions in this case).

A table function can be referenced in an SQL statement anywhere that a table can normally be referenced. The table function, or a nested table expression, is identified in the query by the use of the $TABLE$ keyword as in the following example:

```sql
SELECT TAB1.EMPNO, TAB2.TEMPERATURE, TAB2.FORECAST
FROM EMP, TABLE(WEATHERFUNC(CURRENT DATE)) AS TAB2
```

In the above query, the $TABLE$ keyword was used to identify a nested table expression called $TAB2$ that was an invocation of the table UDF called $WEATHERFUNC$. The query returns the employee number along with some weather information in some of the columns that are returned from the $WEATHERFUNC$ table function. This is a fairly simple invocation of a table function.

More importantly, you can embed correlated references within a nested table expression. While the weather may not be useful information to return with employee data, perhaps retrieving the resume and credit information from an external source is. In this case, we can pass the employee number as a correlated reference into the table expression identified by the $TABLE$ keyword and ultimately pass it into the table UDF:

```sql
SELECT TAB1.EMPNO, TAB2.RESUME, TAB2.CREDITINFO
FROM EMP AS TAB1, TABLE(EMPRPT(TAB1.EMPNO)) AS TAB2
```

The $TABLE$ keyword tells DB2 to look to the left of the keyword when attempting to resolve any otherwise unresolvable correlated references within the table expression. If the join was coded in reverse (that is, the invocation of the $EMPRPT$ table UDF appears in the statement before the $EMP$ table), then the correlated reference to the $TAB1.EMPNO$ column would not have been resolved, and the statement would not have compiled successfully.

The use of the $TABLE$ keyword can be expanded beyond that of correlated references as input into table UDFs. The same keyword can be utilized with a nested table expression that may benefit from a correlated reference. This can be especially useful when the nested expression is performing an aggregation and only needs to work on a subset of the data in the table it is accessing. In the following example, we need to list the employee number and salary of each employee, along with the average salary and head count of all employees in their associated departments. This is traditionally coded as a left outer join of two table expressions, the first getting the employee numbers and salaries, and the second calculating the head count and average salary for all
departments. If there is filtering against the employee table, then the entire table might be unnecessarily read to perform the aggregations:

```sql
SELECT   TAB1.EMPNO, TAB1.SALARY, 
         TAB2.AVGSAL, TAB2.HDCOUNT
FROM
   (SELECT EMPNO, SALARY, WORKDEPT
    FROM   DSN8610.EMP
    WHERE  JOB='SALESREP') AS TAB1
LEFT OUTER JOIN
   (SELECT AVG(SALARY) AS AVGSAL, 
    COUNT(*) AS HDCOUNT, WORKDEPT
    FROM   DSN8610.EMP
    GROUP  BY WORKDEPT) AS TAB2
ON TAB1.WORKDEPT = TAB2.WORKDEPT;
```

Here, the entire EMP table has to be read in the TAB2 nested table expression in order to calculate the average salary and headcount for all departments. This is unfortunate, because we only need the departments that employ sales reps. We can use the TABLE keyword and a correlated reference to TAB1 within the TAB2 expression to perform filtering before the aggregation:

```sql
SELECT   TAB1.EMPNO, TAB1.SALARY, 
         TAB2.AVGSAL, TAB2.HDCOUNT
FROM     DSN8610.EMP TAB1,
         TABLE(SELECT AVG(SALARY) AS AVGSAL, 
               COUNT(*) AS HDCOUNT
               FROM   DSN8610.EMP
               WHERE  WORKDEPT = TAB1.WORKDEPT) AS TAB2
WHERE TAB1.JOB = 'SALESREP';
```

**Polymorphism and UDFs**

DB2 UDFs subscribe to the object-orientated concept of polymorphism. Ad hoc polymorphism (better described as overloading) allows an SQL statement to issue the same function against varying parameter lists and/or data types. This overloading requires us to create a unique definition for each variation of a particular function in data types or number of parameters. Polymorphism basically means "many changes," and for DB2 functions it means that many functions can have the same name. These functions are identified by their signature, which is comprised of the schema name, the function name, the number of parameters, and the data types of the parameters. This enables us to create UDFs for all of our UDTs. These sourced UDFs can assume the same name as the UDFs of built-in functions they are sourced from, but are unique in the system due to the data type of their parameter(s). It also allows us to define SQL or external UDFs to accommodate any variation in data type or number of parameters. For example, if we need a variation of our TOTMON function that accommodates timestamps, we could create the following function:
CREATE FUNCTION TOTMON (STARTX TIMESTAMP, ENDY TIMESTAMP)
RETURNS INTEGER
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
NOT DETERMINISTIC
RETURN ABS( (YEAR(STARTX - ENDY)*12) + MONTH(STARTX - ENDY) );

The only thing that differs between this TOTMON and the original TOTMON is that the input parameters here are TIMESTAMPS instead of DATES. From the application programming point of view, this enables an SQL statement to issue a TOTMON function regardless of whether it is using a pair of dates or timestamps as input. However, it requires that the people responsible for deploying the UDFs do so with consistency of functionality for like-named functions.

External UDF Execution
The external scalar and table UDF programs execute in an OS/390 WLM environment, much in the same way as stored procedures. The WLM environment is supported by one or more WLM address spaces. The WLM keywords that name the WLM environment in which to execute should be specified when creating the function, or else the program defaults to the WLM environment specified at installation time. This default environment can be seen in the SYSIBM.SYSROUTINES catalog table. UDFs execute under the same thread as the invoking program and will run at the same priority, utilizing the WLM enclave processing.

Monitoring and Controlling UDFs

You can invoke UDFs in a SQL statement wherever you can use expressions or built-in functions. UDFs, like stored procedures, run in WLM-established address spaces. DB2 UDFs are controlled by the following commands.

The START FUNCTION SPECIFIC command activates an external function that has been stopped. You cannot start built-in functions or UDFs that are sourced on another function. You can use the START FUNCTION SPECIFIC command to activate all or a specific set of stopped external functions. To activate an external function that is stopped, you would issue the following command:

START FUNCTION SPECIFIC (function-name)

The SCOPE (GROUP) option can also be used on the START FUNCTION command to allow you to start a UDF on all subsystems in a data sharing group.
The DB2 command `DISPLAY FUNCTION SPECIFIC` will display statistics about external UDFs that are accessed by DB2 applications. This command will display an output line for each function that a DB2 application has accessed. The information that is returned by this command will reflect a dynamic status for a point in time and may change before another `DISPLAY` is issued. This command does not display information about built-in functions or UDFs that are sourced on another function. To display statistics about an external UDF accessed by DB2 applications, issue the following command:

```
  DISPLAY FUNCTION SPECIFIC (function-name)
```

**Stopping UDFs**
The DB2 command `STOP FUNCTION SPECIFIC` will prevent DB2 from accepting SQL statements with invocations of the specified functions. This particular command will not prevent SQL statements with invocations of the functions from running if they have already been queued or scheduled by DB2. Built-in functions or UDF that are sourced on another function cannot be explicitly stopped. While the `STOP FUNCTION SPECIFIC` command is in effect, any attempt to execute the stopped functions are queued. You can use the `STOP FUNCTION SPECIFIC` command to stop access to all or a specific set of external functions.

This command stops an external function. Built-in functions or UDFs that are sourced on another function cannot be stopped. Use the `START FUNCTION SPECIFIC` command to activate all or a specific set of stopped external functions.

To prevent DB2 from accepting SQL statements with invocations of the specified functions, issue the following statement:

```
  STOP FUNCTION SPECIFIC (function-name)
```

**UDF Statistics**
The optimizer will use statistics, if available, for estimating the costs for access paths where UDFs are used. The statistics that the optimizer needs can be updated by using the `SYSSTAT.FUNCTIONS` catalog view. There is a field in the statistics report that allows you to view the maximum level of indirect SQL cascading. This includes cascading due to triggers, UDFs, or stored procedures (see Figure 15–4).

**Cost Information**
User-defined table functions will add additional access cost to the execution of an SQL statement. In order for DB2 to determine the cost factor for the use of user-defined table functions in the selection of the best access path for an SQL statement, the total cost of the user-defined table function must be determined. This is determined by three components:
• Initialization cost that results from the first call processing.
• Cost that is associated with acquiring a single row.
• Final call cost that performs the cleanup processing.

To determine the elapsed and CPU time spent for UDF operations, you can view an accounting report (see Figure 15–4).

Catalog Information
The SYSIBM.SYSROUTINES catalog table describes UDFs. To retrieve information about UDFs, you can use the following query:

```sql
SELECT SCHEME, NAME, FUNCTION_TYPE, PARM_COUNT
FROM SYSIBM.SYSROUTINES
WHERE ROUTINETYPE='F'
```

Large Objects

As of v6, DB2 allows us to store large objects. It has new datatypes to handle these objects, and there are several application processing issues associated with handling these new types of data.

LOB Datatypes
Three DB2 data types support LOBs:
• **BLOB** data type (binary large object)
  – Binary strings (not associated to a CCSID)
  – Good for storing image, voice, and sound data

• **CLOB** datatype (character large object)
  – Strings made of single-byte characters or single-/double-byte characters with an associated CCSID
  – Use if data is larger than `VARCHAR` allows

• **DBCLOB** datatype (double-byte character large object)
  – Strings made of double-byte characters with an associated CCSID

Each of these can contain up to 2 gigabytes of data, although in most cases the amount of storage for individual columns will be considerably less (depending on the type of data stored). There is a large use today of the 32KB long `VARCHAR` column, which has limitations in both size and functionality. Most of this use will probably be replaced by using LOBs in the future, certainly for new applications and functions.

**LOB Implementation**

LOBs are implemented with structures that are different than normal tables and tablespaces. A LOB tablespace must be created for each column (or each column of each partition) of a base LOB table. This tablespace contains the auxiliary table, which must have an auxiliary index associated with it. The LOB tablespace has a different recovery scheme, optional logging, and different locking options. For information on LOB locking, refer to Chapter 16, “Locking and Concurrency.”

If a table contains a LOB column and the plan or package is bound with `SQLRULES(STD)`, then DB2 will implicitly create the LOB tablespace, the auxiliary table, and the auxiliary index. DB2 will choose the name and characteristics for these implicitly created objects.

**NOTE:** It is better to develop naming standards beforehand for these objects and control placement of them. This is critical for management and performance.

**INSERTing and LOADing LOBs**

LOB loading and insertion is also different than processes for non-LOB data. The methods are also entirely different depending on whether or not extenders are used. Without extenders, there are some real limitations that need to be addressed when inserting LOB data, primarily the 32KB limit and logging impacts. If the total length of the LOB column and the base table row is less than 32KB, then the `LOAD` utility can insert the LOB column. When the limits of `LOAD` are exceeded, then `SQL INSERT` or `UPDATE` statements need to be used. But the `SQL INSERT` has its own limitations in that enough memory...
needs to be available to hold the entire value of the LOB. The limitations are the amount of memory available to the application and the amount of memory that can be addressed by the language used. If the LOBs are all small, then it is not as much of an issue, as memory and language constructs would be available. But when dealing the very large LOBs, the differences can be seen easily when comparing the C language construct with COBOL.

C language for a LOB host variable,

```sql
SQL TYPE IS CLOB(20000K) my_clob;
```

is generated by DB2 as

```c
struct { unsigned long length;
    Char data[20960000];
} my_clob;
```

COBOL language for a LOB host variable,

```cobol
01 MY-CLOB     USAGE IS SQL TYPE IS CLOB(20000K).
```

is generated by DB2 as

```cobol
01 MY-CLOB.
    02 MY-CLOB-LENGTH   PIC 9(9) COMP.
    02 MY-CLOB-DATA.
        49 FILLER   PIC X(32767).
        49 FILLER   PIC X(32767).
        49 FILLER   PIC X(32767).
```

This is another area where extenders assist in solving the problem. When a table and column are enabled for an extender, the whole process changes. An INSERT statement can be used in the program to contain extender functions (UDFs) that allow the image to be loaded into the database directly from an external file. Actually with the image extender, for example, the image content is inserted into an administrative support table, and another record is then inserted into another administrative table describing the attributes of the image, such as number of colors, thumbnail-sized version, format characteristics (JPEG, TIF, etc.). The extenders require WLM to be installed in support of the extender UDFs and stored procedures, and specifically using WLM in goal mode for performance reasons.

Even though the LOB data is stored in an auxiliary table, the insert statement specifies the base table. You can read the LOB data from a file in your DB2 program and place the data into the declared DB2 LOB variable. The insert statement then simply references the LOB variable. For example, if you wanted to insert employee resumes into the EMP_RESUME table, which includes a 2MB CLOB data type to hold resumes, you declare the resume variable in your program:
Then you populate the resume variable with the CLOB data and perform the insert:

```sql
EXEC SQL INSERT INTO EMP_RESUME VALUES (:EMPNO, :RESUME);
```

DB2 uses contiguous storage in data spaces to store LOBs that your program is manipulating. Because LOBs can be quite large, DB2 avoids materializing LOBs until completely necessary. The amount of storage required depends on the size of the LOBs and the number of LOBs referenced within a statement. The amount of storage required by your program and DB2 can become quite large. For this reason, you can use LOB locators to manipulate LOB data without retrieving that data from the DB2 table. A LOB locator is declared in the application program:

```sql
SQL TYPE IS CLOB_LOCATOR resume_loc;
```

An SQL statement can reference the locator, and the LOB is not materialized in DB2 until absolutely necessary, and it is never moved into the application memory:

```sql
SELECT RESUME
  INTO   :resume_loc
  FROM   EMP_RESUME
  WHERE  EMPNO=:empno;
```

Further SQL statements can reference the locator variable, allowing the LOB to be manipulated in various ways. One way would be by using SQL SET commands. DB2 will manipulate the LOB data within the table, and only materialize the LOB as needed.

**LOBs and UDTs**

Being able to store LOBs and manipulate them through extenders is only part of the story. The capability also exists to define new distinct data types based on the needs of particular applications. A UDT, also known as a distinct type, provides a way to differentiate one LOB from another LOB, even of the same base type, such as BLOB or CLOB. A UDT is not limited to objects and can be created for standard data types as well as LOBs.

Even though stored as LOBs (binary or character), image, video, spatial, XML, and audio objects are treated as types distinct from BLOBs and CLOBs and distinct from each other. For example, suppose an application that processes spatial data features needs a polygon data type. You can create a distinct type named `polygon` for polygon objects, as follows:

```sql
CREATE DISTINCT TYPE polygon AS BLOB (512K)
```
The polygon-type object is treated as a distinct type of object even though internally it is represented as a 512-KB binary object (BLOB). UDTs are used like SQL built-in types to describe the data stored in columns of tables. The extenders create distinct datatypes for the type of object they process, such as image, audio, and video, which makes it easier for applications to incorporate these types of objects.

```
CREATE TABLE DB2_MAG_DEMO
  (GEO_ID CHAR(6),
   EURO_ANNUAL  EURO_DOLLAR,
   OZ_ANNUAL    AUSTRALIAN_DOLLAR,
   US_ANNUAL    US_DOLLAR,
   DEMO_GRAPHIC POLYGON)
```

Casting functions allow operations between different data types—for example, comparing a slide from a video UDT to an expression. You must cast the expression to a UDT (video in this example) in order for the comparison to work. There are casting functions supplied with DB2 (CHAR, INTEGER, and so on), and there are others created automatically whenever a UDT is created with the CREATE DISTINCT TYPE statement.

### Extenders

Extenders for DB2 help in the use of LOBs, the base storage for the object-relational environment. Extenders are a complete package that defines distinct data types and special functions for many types of LOBs, including image, audio, video, text, XML, and spatial objects. This allows us not to worry over defining these data types and functions in applications. You can use SQL to manipulate these data types and functions. LOBs can be from relatively small to extremely large, and can be cumbersome to deal with.

### LOB and Extender Usage

When using an extender for a particular LOB type, there are additional options that actually allow the data to be stored in its native format in separate files, such as a picture that is a single JPEG file. In this example, the hierarchical path name would be stored in support tables that would allow the extender to use this indirect reference to process the actual data. The extenders also require administrative support tables that vary based on the extender used. These tables are also referred to as metadata tables, as their content enables the extenders to appropriately handle user requests, such as inserting audio, displaying images, and so on. These tables identify base tables and columns that are enabled for the extender and reference other support tables used to
hold attribute information about LOB columns. Triggers supplied by the
extenders are used to update many of these support tables when underlying
LOB data is inserted, updated, or deleted. At the present time, there are six
extenders available in the DB2 family: image, audio, video, text, XML, and
spatial. Many others are planned, and vendors also supply extender packages.

Applications generally use SQL to retrieve pointers to the data, and UDFs are
used to assist with more complex and unique operations. Extender APIs will
be more commonly used as all the coding is supplied for dealing with the
LOBs. The extender for image data comes with 18 UDFs; audio has 27 UDFs,
video has 18 UDFs, and the QBIC (Query By Image Content) API has 24
UDFs. For applications, that is a considerable advantage both from a "not-
having-to-program-it" standpoint and easing the pain of the learning curve.
For example, when dealing with the image extenders, there are several
different formats supported. The common ones (BMP, EPS, GIF, JPG, TIF) are
provided, of course, along with over 15 other formats. This means that if I
were browsing through a series of pictures, each LOB picture could be of a
different format, but the program would not have to be aware of this because it
would be taken care of by the extender. The same is true of the text extender.
A user could browse through a series of text documents, one in Microsoft
Word format, another in Word Perfect, and so on. But it is not browsing
documents, playing a video, and streaming audio that represent the most
power. It is the searching ability of the LOB extenders. For example, with the
text extender, searching can be done by soundex, synonym, thesaurus,
proximity, linguistic, and several other criteria. With images, the QBIC API is
supplied to allow searching by image content, and this is a very extensive and
powerful API.

Application programming for objects will generally require the use of the
extenders. Without their use, there is basically little that can be done without
extensive user programming. The power of objects comes with the UDFs and
API libraries that are packaged with the extenders. They allow an application
to easily store, access, and manipulate any of the supported object types.
Although the current list supplied is only the six extenders previously
mentioned, many others are in development and will be released in the future
as they are completed. The application programmer will be able to use UDFs
in the SQL to position to the necessary LOB and then use an API to
manipulate it, such as to display a picture on the screen. So, there are really
two completely different libraries to strategize from. As a simple example,
without forcing it to match any particular programming language, the following
represents first storing a picture in a LOB and then displaying it on the screen.
First, insert the data into the LOB by using the DB2IMAGE extender:

```sql
EXEC SQL BEGIN DECLARE SECTION;
    storage_type;
EXEC SQL END DECLARE SECTION;
```
SET storage_type = MMDB_STORAGE_TYPE_INTERNAL

EXEC SQL INSERT INTO MY_PERSONAL_DIGITAL_PICTURES
VALUES ('OZ TRIP 2',
'Sydney Opera House',
DB2IMAGE {
    CURRENT SERVER,
'c:/My Pictures/1999/Australia/OpraHse.jpg',
'ASIS',
:storage_type});

Second, retrieve and display the data on the screen using API DBiBROWSE:

EXEC SQL BEGIN DECLARE SECTION;
    image_handler;
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT PICTURE INTO :image_handler
WHERE NAME = 'Sydney Opera House';
Set return_code to DBiBROWSE("ib %s,
    MMDB_PLAY_HANDLE,
    image_handler,
    MMDB_PLAY_BI_WAIT");

From the pseudocode, it is easy to see that the extenders offer significant power and enable applications to be written quickly. In addition, most of the work is going on at the server, and the client is simply the recipient of all that power. When implementing extenders, keep in mind that the program will need to have enough memory available to support the use of LOBs on GUI clients.

Enabling Extenders

A Software Developers Kit (SDK) and a client and server runtime environment are provided with the DB2 extenders installation package. DB2 extender applications can be executed in a server machine that has the extender client runtime code (automatically installed when the server runtime code is installed) and server runtime code. Extender applications can also be run on a client machine with the client runtime code, and you must ensure that a connection can be made to the server.

Extenders are available in DB2 Version 6 client/server environment. OS/390 is the supported server, and clients can be Windows NT, Windows 98, Solaris, AIX, or OS/390.

When storing image, audio, video, or text objects, you do not store the object in the user table but instead use an extender-created character string referred to as a handle that represents the objects, and the handle will be stored in the
user table. The object is actually stored in an administrative support table (or file identifier if the content is a file). The attributes and handles are also stored in these administrative tables. The extender then links the handle in the user table to the object and its attributes in the administrative tables.

Text Extenders

Text extenders bring full-text retrieval to SQL queries for searching large text documents intelligently. With the use of text extenders, you can search several thousand large text documents very quickly. You can also search based upon word variations and synonyms. These documents can be stored directly in the database or in a separate file.

Files such as native word-processing documents can be search by keywords, wildcards, phrases, and proximity. IBM has built into these text extenders a high-performance linguistic search technology, giving you multiple options for searching and retrieving documents. These text searches can be integrated with your normal SQL queries. This enables you to integrate into your SELECT statements the ability to perform attribute and full-text searches very easily.

The following example shows how to perform this integration. In this example, we perform a SELECT from a table that also performs a search on a specified document using a text extender called DB2TX.CONTAINS. We are searching the legal cases document to see if the words “malpractice” and “won” appear in the same paragraph for cases occurring after 1990-01-01. LEGCSE_HANDLE refers to the column LEGCSE that contains the text document.

```
SELECT DOC_NUM, DOC_DATE
FROM LEGALCLAIMS
WHERE DOC_DATE > '1990-01-01'
AND DB2TX.CONTAINS
    (LEGCVE_HANDLE,
    "malpractice"
    IN SAME PARAGRAPH AS "won") = 1
```

Text extenders allow applications to

- Search documents of several languages and formats.
- Perform wildcard searches using masks.
- Perform a search for words that sound like the search input.
- Perform fuzzy searches for like words (various spellings).
- Perform searches for specific text, synonyms, phrases, and proximity.
- Perform free-text searches where the input is a natural language.
Indexing Text Extenders
Scans are just as undesirable in text documents as they are with our DB2 tables. We need to create indexes so that sequential scans of documents are not necessary. By using a text index, you can speed up the searches performed on these documents.

A text index contains important words as well as a list of words known as stop words, such as and and the, which will not be in a text index. This list can be modified, but you would only want to do it once at installation time. When a request is made, the index is searched for the important terms to determine which documents contain those specified terms.

To set up a text index, you first record the text documents that need to be indexed in a log table. This process occurs when a DB2 trigger is fired off during an insert, update, or delete of a column of a text document. Then, when the terms are inserted or updated in the text document, they are added to the index. They are also deleted from the index if they are deleted from the text document.

There are four types of text indexes, and the type must be established before you implement columns that will be using text extenders. Not all search options are available by all index types, so you want to make sure the index will suit your criteria for searching. The four types of indexes are as follows:

- **Linguistic:** In this type of index linguistic processing is performed during the analysis for the text when creating an index. Before a word is inserted into the index, it is reduced to its base form. Queries also use linguistic processing when searching against this index. This index requires the least amount of space, but searches may be longer than those done against a precise index.

- **Precise:** In this index the search terms are exactly as they are in the text document and are case-sensitive. The same processing is used for the query search terms, so they must match exactly. The search can be broadened by using masks. This index provides a more precise search, and the retrieval and indexing is fast, but more space is required for its storage.

- **Dual:** Dual indexes are combinations of linguistic and precise indexes. This allows the user to decide which type of search to use. This index requires the most amount of disk space. It is slower for searching and indexing than the linguistic indexes and is not recommended for a large number of text documents.

- **Ngram:** The Ngram index is used primarily for indexing DBCS documents; it analyzes text by parsing sets of characters. This index type also supports fuzzy searches.

When creating tables that will support the ability to search text using extenders, you must consider a few design options. You can create one text
You can also have multiple indexes on a single text column. You may want to do this if you need the ability to allow different types of searches on a text column. And just like other DB2 indexes, these indexes will need to be reorganized too. If you have a text column that is continually updated, you will need to reorganize it. However, when using these indexes, the text extender automatically reorganizes them in the background. Despite this feature, you still may have to reorganize an index manually every so often, depending on its volatility. This is done with the \texttt{REORGANIZE INDEX} command. Issue the \texttt{GET INDEX STATUS} command to see if an index needs reorganization.

\textbf{Frequency of Index Updates}

When text documents are added, deleted, or changed, their content must be synchronized with the index. This information is automatically stored by triggers in a log table, and the documents will be indexed the next time an index update is executed.

The indexes can be immediately updated via the \texttt{UPDATE INDEX} command, but it is easier to have this performed automatically on a periodic basis. This time-based information is kept in an environment variable called \texttt{DB2TXUPDATEFREQ}, which provides default settings that can be changed during the \texttt{ENABLE TEXT COLUMN} or \texttt{ENABLE TEXT TABLE} commands. For an existing index, you can use the \texttt{CHANGE INDEX SETTINGS} command to change the variable settings.

The variable for determining when indexing should occur is based on the minimum number of queued text documents in the log table, and when this minimum is reached, the index is updated. Because updating indexes is a very resource-intensive and time-consuming task, this frequency should be set carefully.

\textbf{Catalog View for Text Extenders}

There is a catalog view created for each subsystem when you run the \texttt{ENABLE SERVER}. This view is \texttt{DB2TX.TEXTINDEXES}. It has information about the tables and the columns that have been enabled for the text extender. The entries are made during table, column, or external file enablement. If they are disabled, the row is removed. You can view the entries in the catalog view via SQL. In this view you can see such information as how often the indexes are
scheduled for updates, whether or not you have a multiple-index table, and the type of index.

**Image, Audio, and Video Extenders**

The DB2 video extender can store as many as three representative frames per shot. By displaying the frames, you get a quick yet effective view of a video's content. The DB2 video extender provides sample programs that demonstrate how to build and display a video storyboard.

Video storyboards allow you to preview videos before you download and view them. This can save you time and reduce video traffic on the network. When image data is placed into a table using the `DB2IMAGE` UDFs, many processes are performed for the application automatically. The following code demonstrates using this function.

```sql
EXEC SQL INSERT INTO CONSULTANTS VALUES(
  :cons_id,
  :cons_name,
  DB2IMAGE(
    CURRENT SERVER,
    '/RYC/images/current.bmp'
    'ASIS',
    MMDB_STORAGE_TYPE_INTERNAL,
    :cons_picture_tag);
```

In this particular example, the `DB2IMAGE` reads all the attributes about the image (height, width, colors, layers, pixels, and more) from the source image file header, in this case the `current.bmp`. All the input is of a standard supported format, and all graphic files contain header information about the structure of the content. The function then creates a unique handle for the image and records all the information in the support table for administrative control of the image. This table contains:

- The handle for the image
- A timestamp
- The image size in bytes
- The comment contained in :cons_picture_tag
- The content of the image

The content of the image source file is inserted into the LOB table as a BLOB. There is no conversion done, and the image is stored in its native format. There is a record in the administrative table that contains all the image-specific attributes, such as the number of colors in the image, as well as a thumbnail-sized version of the image.
This example uses the storage type constants. We used `MMDB_STORAGE_TYPE_INTERNAL` to store the image into a database table as a BLOB. By using the extenders, we could have stored it elsewhere. If you want to store the object and have its content remain in the original file on the server, you can specify the constant `MMDB_STORAGE_TYPE_EXTERNAL`. Just because you are using LOBs does not mean that they have to be in DB2-managed tables. The administrative support table for image extenders tells where the LOB is actually stored. This does require Open Edition support services on the OS/390. From a performance perspective, there are many considerations as to where the LOB is stored, how it is used, where it is materialized, and so on.

**XML Extenders**

XML has been added to the list of available extenders. For the next generation B2B e-commerce solutions, XML is the standard for data interchange. With the XML extender for DB2, you will be able to leverage your critical business information in DB2 databases in order to engage in B2B solutions using XML-based interchange formats.

In terms of Web publishing, you can use XML documents stored in DB2 in a single column or as a collection of data items in multiple columns and tables. The text extender in DB2 supports structured documents like XML. The powerful search functions provided can now be applied to a section or a list of sections within a set of XML documents. This can significantly improve the effectiveness of the search. Additionally, specific XML elements or attributes can be automatically extracted into traditional SQL data types to leverage DB2’s sophisticated indexing and SQL query capabilities. The DB2 XML extender also supplies a visual administration tool for easy definitions for mapping elements and attributes from an XML document into columns and tables.

**Summary**

In this chapter we talked about very powerful DB2 and SQL features, such as UDFs and triggers. These features can reduce the amount of application development time and maintenance. We can use these features to encapsulate our code into one place for use by many applications.

DB2 has implemented some object-relational functions, such as LOB support, UDTs, and UDFs. These features also allow us great flexibility and power in our applications. LOBs, of course, give us the ability to store large amounts and different types of data, such as multimedia, giving us the ability to develop very sophisticated applications.
Additional Resources

IBM DB2 V7 for OS/390 Image, Audio, and Video Extenders Administration and Programming – SES1-2199-0

IBM DB2 V7 for OS/390 XML Extender Administration and Programming – SES1-2201-00

IBM DB2 V7 for OS/390 Text Extender Administration and Programming – SES1-2200-00