Chapter 3.
"Objects and Types"
In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter 3 “Objects and Types”
A synopsis of the book’s content
Information on where to buy this book

About the Author

Pawan Kumar is a Technical Architect with current expertise in Enterprise Content Management with EMC Documentum. His expertise spans solution architecture, document management, system integration, web content management, business process management, imaging and input management, and custom application development. Pawan has experience developing products as well as delivering business solutions on the Documentum platform. He is intimately familiar with effective processes and tools for achieving business objectives through Documentum-based technology solutions. He has led and executed requirements and design workshops, architecture design, scoping, estimation, project planning, resource planning, technical design, software development, software testing, solution roll-out, and ongoing support for the deployed solutions. He has also created two products for the Documentum platform.

Pawan has been architecting, designing, and developing enterprise applications for ten years. He has developed software systems for financial services, healthcare, pharmaceutical, logistics, energy services, and retail industries. Pawan has a BS in Electrical Engineering from the Indian Institute of Technology, New Delhi (India) and MS in Computer Science from the University of North Carolina at Chapel Hill.

Currently, Pawan provides consulting and training services through doQuent (http://doquent.com), which was founded with the vision of enabling client success in content-related business initiatives. He also believes in giving back to the community. He founded the free online Documentum community—dm_cram (http://dmcram.org), which is a test preparation resource for Documentum exams. He is also an active contributor to the Documentum-users Yahoo! User group, where Documentum community members seek help for their technical challenges. He can be reached at pk@doquent.com.

EMC Documentum is the leading enterprise content management (ECM) platform globally. EMC Proven Professional certification is an exam-based certification program, which introduced a new EMC Proven Content Management Application Developer (EMCAD) track in early 2007. The first exam in this track is Content Management Foundations Associate-level Exam, whose exam code is E20-120.

This book is a complete study guide and includes study material and practice questions to prepare for this exam. Even though this book focuses on certification preparation, it strives to serve Documentum beginners and practitioners irrespective of their interest in the certification exam. It can also serve as a handy guide and quick reference to the technical fundamentals that is fully up to date for Documentum 5.3. Beginners are introduced to concepts in a logical manner while practitioners can use it as a reference to jump to relevant concepts directly.

**What This Book Covers**

This book is organized in chapters based on the structure of the recommended training for the CMF exam.


The chapters are grouped together in parts to provide a logical grouping and order of topics as described below.

**Part 1: Fundamentals** (Chapters 1 - 4)

*ECM Basics* introduces the basic concepts of content management. *Working with Content* describes the aspects of creating and manipulating content. *Objects and Types* lays the foundation of designing and using metadata. *Architecture* describes the key components of the EMC Documentum platform and how they interact to provide the content management capabilities.

Part 2: Security (Chapters 5 - 7)
Users and Privileges describes the core concepts related to users for implementing security in Documentum. Groups and Roles provides additional capabilities for facilitating security management for groups of users. Object Security introduces permissions and ties them to users, groups, and privileges to realize the security model.

Part 3: User Interface (Chapter 8)
Searching describes the features for finding relevant content stored in a repository. While other user interface aspects are covered throughout the book, searching is described separately because of its fundamental importance to content management.

Part 4: Application Development (Chapters 9 - 12)
Custom Types describes how to create user-defined metadata structures and fundamental customization aspects. DocApps describes how to package the development artefacts for reuse and portability across repositories. Workflows and Lifecycles describe how to model and implement business processes in Documentum.

Part 5: Advanced Concepts (Chapters 13 - 14)
Aliases describe a mechanism for dynamic assignment of ownership, locations, and permissions. Virtual Documents describes how multiple documents can be managed as one larger document to facilitate collaboration.

There are two practice tests at the end of this book.

There is a set of questions at the end of each chapter. These questions are meant to test your understanding. A good way to prioritize and focus your efforts is to use the questions to identify the areas where you score low and then work on those areas.

The answers to all the questions and the solutions to the practice tests have been provided at the end.

3

Objects and Types

In this chapter, we will explore the following concepts:

- Objects and types
- Type hierarchies
- Object persistence
- Querying objects

Objects

Documentum uses an object-oriented model to store information within the repository. Everything stored in the repository participates in this object model in some way. For example, a user, a document, and a folder are all represented as objects. An object stores data in its properties and has methods that can be used to interact with the object.

A content item stored in the repository has an associated object to store its metadata. For example, a document stored in the repository may have its title, subject, and keywords stored in the associated object. However, note that objects can exist in the repository without an associated content item. Such objects are sometimes referred to as contentless objects. For example, a user object or a permission set object does not have any associated content.

Note that the term method may be used in two different contexts within Documentum. A method as a defined operation on a type is usually invoked programmatically through DFC. There is also the concept of a method representing code that can be invoked via a job, workflow activity, or a lifecycle operation. This qualification will be made explicit when the context is not clear.

Each object property has a data type, which can be boolean, integer, string, double, time, or ID. A boolean value is true or false. A string value consists of text. A double value is a floating point number. A time value represents a timestamp, including dates. An ID value represents an object ID that uniquely identifies an object in the repository.

A property can be single-valued or repeating. Each single-valued property holds one value. For example, the object_name property of a document contains one value and it is of type string. This means that the document can only have one name. On the other hand, keywords is a repeating property and can have multiple string values. In this example, a document can have object_name='invoice.pdf' and keywords='invoice.pdf', 'ABC Corp.', 'Trading'.

The following figure shows a visual representation of this object. Typically, only properties are shown on the object while methods are shown when needed.

r_object_id is a special property of every persistent object. It is used to uniquely identify an object and encodes some information within the property itself. It is a 16-character string value where each character is a hex (hexadecimal) digit. The first two digits constitute a tag representing the type of the object.

For example, 09 means that the object has a type that is dm_document or its subtype—the object represents a document rather than a user, group, or something else. Subtypes are explained later in this chapter. The next 6 digits represent the repository ID—a numeric identifier assigned to the repository. The last 8 digits represent a unique ID within the repository and this ID is generated by the Content Server.

Note that EMC Documentum assigns a unique range of repository IDs to each of its customers for the various repositories served by their Content Server installations. As long as these assigned repository IDs are used uniquely, r_object_id will uniquely identify an object across all repositories.

Methods are operations that can be performed on an object. An operation often alters some properties of the object. For example, the checkout method can be used to check out an object. Checking out an object sets the \texttt{r_lock_owner} property with the name of the user performing the checkout. Methods are usually invoked using Documentum Foundation Classes (DFCs) programmatically, though they can be indirectly invoked using DQL and API.

### Object Types

Different objects may represent different kind of entities—one object may represent a workflow while another object may represent a document, for example. As a result, these objects may have different properties and methods. Each time an object is created in the repository, it needs to be determined what properties and methods it is going to have. This information comes from an \textbf{object type}.

An object type is a template for creating objects. In other words, an object is an \textit{instance} of its type. A Documentum repository contains many predefined types and allows addition of new user-defined types (also known as \textit{custom} types). User-defined types offer important capabilities and are described in detail in a separate chapter—\textit{Custom Types}.

The most commonly used predefined object type for storing documents in the repository is \texttt{dm_document}. Objects in a repository can be organized using \textit{folders}, which are stored as objects of type \texttt{dm_folder}. The root folder in a folder tree is called a \textit{cabinet} and is stored as an object of type \texttt{dm_cabinet}. Users are represented as objects of type \texttt{dm_user} in the repository. A group of users is represented as an object of \texttt{dm_group}. Workflows use a process definition object of type \texttt{dm_process}, while the definition of a lifecycle is stored in an object of type \texttt{dm_policy}. These object types are described in more detail at various places in later chapters.
Here is a figure displaying the object types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dm_sysobject</strong></td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>object_name</td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>title</td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>subject</td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td>authors</td>
<td><img src="image5" alt="Diagram" /></td>
</tr>
<tr>
<td>...</td>
<td><img src="image6" alt="Diagram" /></td>
</tr>
</tbody>
</table>

| **dm_process**      | ![Diagram](image7) |
| package_control     | ![Diagram](image8) |
| r_act_def_id        | ![Diagram](image9) |
| r_act_name          | ![Diagram](image10) |
| r_act_priority      | ![Diagram](image11) |
| ...                 | ![Diagram](image12) |

| **dm_policy**       | ![Diagram](image13) |
| action_object_id    | ![Diagram](image14) |
| allow_attach        | ![Diagram](image15) |
| allow_demote        | ![Diagram](image16) |
| allow_schedule      | ![Diagram](image17) |
| ...                 | ![Diagram](image18) |

While the obvious use of a type is to define the structure and behavior of one kind of objects, there is another very important application of types. A type can be used to refer to all the objects of that type as a set. For example, queries restrict the scope of search by specifying a type and as a result only the objects of that type are considered for the results. Queries are introduced later in this chapter.

As another example, audit events can be restricted to a particular object type resulting in only the objects of this type being audited. Auditing is described in more detail in *User and Privileges* (Chapter 5).

Type Names and Property Names

Each object type uses an internal type name, such as dm_document, which is used for uniquely identifying the type within queries and application code. Each type also has a label, which is a user-friendly name often used by applications for displaying information to the end users. For example, the type dm_document has the label Document. Conventionally, internal names of predefined (defined by Documentum) types start with dm_.

Just like an object type each property also has an internal name and a label. For example, the label for property object_name is Name. There are some additional conventions for internal names for properties. These names may begin with the following prefixes:

1. r_: (read only) This prefix normally indicates that the property is controlled by the Content Server and cannot be modified by users or applications. For example, r_object_id represents the unique ID for the object. On the other hand, r_version_label is an interesting property. It is a repeating property and has at least one value supplied by the Content Server, while others may be supplied by users or applications.

2. i_: (internal) This prefix is similar to r_ except that this property is used internally by the Content Server and normally not seen by users and applications. As discussed in the last chapter, i_chronicle_id binds all the versions in a version tree together and is managed by the Content Server.

3. a_: (application) This prefix indicates that this property is intended to be used by applications and can be modified by applications and users. For example, the format of a document is stored in a_content_type. This property helps Webtop launch an appropriate desktop application to open a document. The other three prefixes can also be considered to imply system or non-application attributes, in general.

4. _: (computed) this prefix indicates that this property is not stored in the repository and is computed by Content Server as needed. These properties are also normally read-only for applications. For example, each object has a property called _changed, which indicates whether it has been changed since it was last saved. Many of the computed properties are related to security and most are used for caching information in user sessions.

Type Hierarchy

It is common for different types to be related in some sense and share properties and methods. In true object-oriented style, Documentum allows persistent types to be organized in an inheritance-based type hierarchy. A type can have one supertype and inherit all the supertype properties as its own. The complete set of properties belonging to a type is the union of the inherited properties and properties explicitly defined for that type. In this relationship, the new type is called a subtype.

The super and sub prefixes are based on the visual representation of this relationship where the supertype is positioned logically higher than the subtype, as shown in the following figure:

![Diagram of type hierarchy]

Note that supertype and subtype are relative terms. This means that when using either of these terms we refer to two types. A type can be a subtype for one type and supertype for another type at the same time. When many of these related relationships are visually represented together, they create a structure similar to an inverted tree (root at the top) known as a type hierarchy. Readers familiar with object-oriented modeling will recognize this type hierarchy as a class-inheritance hierarchy. The following figure shows a portion of the type hierarchy for the predefined Documentum types:

![Diagram of type hierarchy with predefined types]

dm_document is an important type since it is almost always involved with document storage and that is a key capability of the Documentum platform. It is an interesting type because it has no properties of its own and it inherits all its properties from dm_sysobject.

One may question the point of having a separate type without any properties of its own. Remember the comment about using a type for treating the objects of that type as a set? dm_document as a separate type enables us to refer to all the objects of this type and subtypes as a set. It can also be used for the complementary set, for example, identifying all the objects of type dm_sysobject but not of the type dm_document.

### Object Persistence

Objects that are stored in the repository are called **persistent** objects and their types are referred to as persistent types. All persistent types are part of a type hierarchy rooted in the internal type **persistent object**, which has the following properties:

1. `r_object_id`: This is used for unique identification, assigned to the object by the Content Server. This property is described earlier in this chapter.
2. `i_vstamp`: This is used internally for version control; it holds the number of committed transactions that have altered this object.
3. `i_is_replica`: This is used in replication and determines whether an object is a **replica** of another in a different repository. Object replication replicates (copies) objects, both content and metadata, from a source repository to a target repository. The object copies in the target repository are known as replica objects.

Objects are stored in the repository using **object-relational technology** where properties are stored in (relational) database tables. Each persistent type is represented by two tables in the repository database—one for storing the single-valued properties and the other for storing the repeating properties. Single-valued properties for a type are stored in a table named `type_name_s`, while repeating properties are stored in a table named `type_name_r`.

For both single-valued and repeating properties, the property names map to the column names in the tables. Further, all of the `_s` and `_r` tables also have a column named `r_object_id`. The `r_object_id` column is used to join the single-valued and repeating properties along with the inherited properties to bring all the properties of an object together.

---

The structure of the _r tables is worth paying extra attention to. Each object can have multiple rows in the _r table where each column represents one repeating property. Usually, two repeating properties of an object are not related to each other. For example, authors and i_folder_id are two repeating properties of dm_sysobject and there is no relation between an author and the ID of a folder that the object is linked to. Yet, these two values may be present in the same record in dm_sysobject_r.

This storage scheme of shared records lets us determine the number of records for an object in its _r table. It is equal to the maximum number of values in any of the repeating properties that is not an inherited property for the object's type.

The following figure illustrates persistence for an object of a custom type my_course:

```
<table>
<thead>
<tr>
<th>course_id</th>
<th>credits</th>
<th>r_object_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS100</td>
<td>3</td>
<td>090006480001126</td>
</tr>
</tbody>
</table>
```

Note that the tables used for persisting objects of a particular type only store the properties explicitly defined for that type. Inherited properties are stored in the tables for the supertypes where they belong. Since dm_document does not have any properties of its own, there is no table named dm_document_s or dm_document_r in the repository database.

It is useful to know how properties are stored in database tables but all the properties of objects can be queried together using DQL without any reference to these tables. Internally, the Content Server uses database views that join appropriate tables to retrieve all the needed properties of the type together.

While most of the types represent persistent objects, there are some types whose objects are used for temporarily storing information in memory. These objects are not stored in the repository and are called non-persistent objects. For example, collection objects are used to store query results and they reside only in memory at run time.

Querying Objects

Document Query Language (DQL) is a query language for Documentum just as Structured Query Language (SQL) is a query language for databases. In fact, DQL is a superset of ANSI SQL, which means that a valid query in ANSI SQL is also a valid DQL query. DQL queries can be executed using IDQL (Interactive DQL shell), Documentum Administrator, Webtop, or programmatically through DFC applications.

DFC provides a rich set of functionality for interacting with objects, including creating, querying, and modifying objects. DFC is a programmatic means of interacting with objects and is used in applications. DQL is used both for scripting and with DFC in applications. In this section, we will examine some DQL queries used for manipulating objects. However, this is just a small overview of DQL capabilities and the DQL Reference Documentation should be used to explore the full set of DQL capabilities.

SELECT Query

A DQL query can be used to inspect or affect one or more objects in a repository. The most common type of DQL query is the SELECT query, which retrieves the properties of one or more objects. For example, consider the following query:

```
SELECT r_object_id, r_creation_date
FROM dm_document
WHERE object_name = 'mydoc.txt'
```

This query shows three keywords—SELECT, FROM, and WHERE. These keywords divide up the query into three parts:

1. **SELECT clause (selected values list)**: The selected values list specifies the properties to be retrieved.
2. **FROM clause**: The FROM clause specifies the object types to be queried.
3. **WHERE clause**: The WHERE clause is optional and specifies the conditions for the objects to meet whose properties will be returned by the query. When the WHERE clause is present, the query is also called a conditional query.

A DQL query can also directly query database tables, though the tables need to be registered first. A registered table is a table from the underlying database that has been registered with the repository. This registration allows the table to be specified in a DQL query, either by itself or in conjunction with a type. A registered table can be used as an object type and its columns can be used as properties.

Now, let's try to understand the semantics of this query. The FROM clause specifies that we want to consider objects of type dm_document. Among these objects, we only want to look at objects that have 'mydoc.txt' in their object_name property. The query will return the object ID (r_object_id property) and creation date (r_creation_date property) for all the resulting objects.

No matter how (DFC or DQL) objects are queried, Content Server always enforces the configured security. Content Server will not return all documents just because a query requests all documents. It will only return the documents that the currently authenticated user is allowed to retrieve.

The same rules apply to the operations other than querying. Repository security is discussed in more detail in later chapters.

Basics

The comma-separated list after SELECT identifies the values to be returned. These values typically come from object properties, though they may include constants and calculations on properties as well. The allowed properties depend on the types specified in the FROM clause. For example:

```
SELECT object_name, title
FROM dm_document
```

Here the selected values are the properties object_name and title for the type dm_document. It is possible to rename the values being returned using the following syntax:

```
SELECT object_name AS Name, title AS Title
FROM dm_document
```

This capability is more useful and desirable when multiple types are present in the FROM clause:

```
SELECT d.r_object_id AS ObjectId, f.r_object_id AS FolderId
FROM dm_document d, dm_folder f
WHERE ...
```

Note that the selected values are both r_object_id, so renaming enables us to distinguish between them. Also note that we need to associate the property name with the type name in this case and it is done by using the prefixes d. and f., where d and f are aliases (unrelated to the aliases in alias sets to be discussed in later chapters) for the types in the FROM clause. It is a good practice to use aliases for types and prefix them to property names when multiple types are present in the FROM clause.

It is rare to run a select query without a WHERE clause because it will return all objects of the specified type(s). The WHERE clause enables us to provide conditions or search criteria and narrow down the search scope to find the specific objects we are looking for.

WHERE Clause

The WHERE clause specifies a condition, which may consist of multiple conditions that an object must satisfy to be a part of the result set. An object participates in the conditions via its properties. Functions, expressions, logical operations, and literals are used along with the properties to define the condition. Some examples below illustrate the usage of WHERE clause.

The following example shows the use of a string literal in the WHERE clause. Note that a string literal is placed within single quotes:

```sql
SELECT object_name
FROM dm_document
WHERE title = 'CS100'
```

The following example shows that a numeric value does not use quotes. This query retrieves objects that have been updated at least once:

```sql
SELECT object_name
FROM dm_document
WHERE i_vstamp > 0
```

An object ID literal is placed within single-quotes. The following query retrieves one specific object from the repository using its object ID:

```sql
SELECT object_name
FROM dm_document
WHERE r_object_id = '0900006480001126'
```

A repeating property in a WHERE clause is typically used with the keyword ANY, as shown in the next example. This query retrieves all documents that have any of the keywords set to invoice:

```sql
SELECT object_name
FROM dm_document
WHERE ANY keywords = 'invoice'
```

Another commonly used condition relates to dates and the DATE function is useful for such situations. The following query retrieves objects that have not been modified since 12/10/2006:

```
SELECT object_name 
FROM dm_document 
WHERE r_modify_date < DATE('12/10/2006')
```

Next we look at UPDATE queries, which are used for modifying objects.

**UPDATE Query**

An UPDATE query updates one or more objects and has the following syntax:

```
UPDATE <type_name> OBJECT 
<property_updates> 
WHERE <condition>
```

The WHERE clause works just as in the SELECT query. As before, the WHERE clause is optional but it is highly recommended that the WHERE clause should not be omitted as far as possible. <type_name> is the type or an ancestor type (supertype or supertype's supertype, and so on) of the object(s) to be updated. <property_updates> specify the property names and the corresponding values to be set. The following example illustrates these concepts:

```
UPDATE dm_document OBJECT 
SET object_name = 'mydoc.txt', 
SET title = 'John''s Document', 
SET authors[0] = 'John', 
SET authors[1] = 'Jane' 
WHERE r_object_id = '0900006480001126'
```

This query shows several new features. Note that the keyword OBJECT (OBJECTS is also acceptable) is required, since we are trying to update the objects. If OBJECT is omitted, the query will attempt to modify the type (rather than objects). <property_updates> is specified using the format SET <property_name> = <value>. If multiple properties are being updated they are separated using commas.

Another point to note is that if a repeating property, like authors in this example, needs to be updated, each individual value needs to be set using this format—SET <property_name>[<index>] = <value>. <index> specifies the position in the list of repeating values for the property and the positions start with 0. Also note that for title we used two apostrophes where we needed one in the value. It is true for all DQL queries that an apostrophe inside a string literal should be replaced with two to escape the special meaning of the apostrophe.

It is not common to distinguish between a supertype and an ancestor type. Often, the term supertype is intended to mean "supertype, supertype's supertype, and so on". The term subtype is also loosely used in a similar fashion to include the descendants in the type hierarchy.

DELETE Query

A DELETE query is similar to an UPDATE query except that there are no properties to be set. A DELETE query has the following format:

```
DELETE <type_name> OBJECT
WHERE <condition>
```

This query does not have many new features. In fact it is probably one of the simplest DQL queries. Again, the WHERE clause is optional but omitting it will result in all objects of the specified type and its subtypes being deleted. You need to be very careful when using DELETE queries. Let's look at an example of the DELETE query:

```
DELETE dm_document OBJECT
WHERE owner_name = USER
AND FOLDER('/Temp')
```

This query deletes all objects of type dm_document or any of its subtypes that are owned by the currently authenticated user and linked to the folder path /Temp.

Note the use of the keyword USER—it gets dynamically replaced with the currently authenticated user when the query is executed. Similarly, TODAY is a keyword that gets replaced with the date on which the query is executed. Some other useful keywords are YESTERDAY, TOMORROW, and NOW. These keywords are used in queries that utilize date or time values.

Further, note the use of keyword AND—it enables conjunction of two conditions in the WHERE clause. OR and NOT can also be used in a similar manner.

A path within a repository is represented in a way similar to a path on the file system. For example, /Temp/mydocs/resume.doc is a path in the repository to a document named resume.doc. This document is linked to a folder named mydoc, which in turn is linked to a cabinet named Temp.

The top-level folders are special and are called cabinets. They always appear as the first element in a path. Each repository has some cabinets created for use by Documentum software. These cabinets are called system cabinets. Temp is a system cabinet, which is frequently used for organizing temporary objects.

The query also illustrates how to search certain folders for objects. The folder predicate can specify one or more folder paths and whether the subfolders of those folders should be included in the search recursively. Consider the modified version of this query:

```
DELETE dm_document OBJECT
WHERE owner_name = USER
AND FOLDER('/Temp/a','/Temp/b',DESCEND)
```

This query deletes all objects of type `dm_document` or any of its subtypes that are owned by the currently authenticated user and linked to the folder path `/Temp/a` or `/Temp/b` or any subfolders of these paths. Note that multiple folders can be specified in the folder predicate and, optionally, `DESCEND` specifies that the subfolders should be included.

### API

API methods can be issued via IAPI or Documentum Administrator in addition to programmatic access through DFC. IAPI can send individual method calls to the server. The API can be used to create scripts for administrative or development purposes. One of the most common uses of the API is to dump an object to view all of its properties. For example, the following API command prints the names and values for all the properties of the object identified by the given object ID:

```
dump,c,'0900006480001126'
```

The API will not be discussed in detail in this book. For exploring the API in detail, please see the *API Reference Documentation*.

### Documentum Product Notes

DQL queries can be executed through IDQL, Documentum Administrator, or Webtop. They can also be executed programmatically using DFC.

API queries can be executed using IAPI, Documentum Administrator, and DFC.

### Checkpoint

At this point you should be able to answer the following key questions:

1. What is the difference between objects and types? How are objects related to types?
2. What information is encoded in the `r_object_id` attribute?

3. What is a type hierarchy? How are objects persisted in the repository database?
4. What are the various ways of querying the objects in a repository? What are some common DQL queries?

**Test Your Understanding**

1. An object can inherit properties and methods from another object (True/False).
2. There is no dm_folder_s table (True/False).
3. Administrators can use DQL to query the objects that DFC would prevent them from accessing due to permission restrictions (True/False).
4. Suppose a custom type my_document is a subtype of dm_document. When an object of type my_document is created, the first two hex digits of r_object_id for this object will be _____.
5. The prefix i_ for predefined properties normally indicates:
   a. Immutable
   b. Internal
   c. Imported
   d. None of the above
6. Suppose a document has only three of its repeating properties set:
   authors='John','Jane';
   keywords='invoice','corporate','finance','software';
   r_version_label='1.2'.

   Given that all of these properties are present in dm_sysobject_r, how many records will this object have in this table?
   a. 1
   b. 2
   c. 4
   d. 8
7. The dm_document_r table stores the authors property since authors is a repeating property (True/False).

8. The first two hex digits in r_object_id represent:
   a. Repository ID
   b. Unique ID within the repository
   c. Object format
   d. None of the above

9. DQL can be used to query databases directly (True/False).

10. The following query will include a dm_document object named mydoc.txt in the result set as long as its permissions allow it (True/False):

    ```sql
    SELECT r_object_id, title
    FROM dm_sysobject
    WHERE object_name = 'mydoc.txt'
    ```
Where to buy this book


Free shipping to the US, UK, Europe, Australia, New Zealand and India.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.