Objects and Types

In this chapter, we will explore the following concepts:

- Objects and types
- Type hierarchies and type categories
- Object and content persistence
- Lightweight and shareable object types
- Aspects
- 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 (also known as attributes) and has methods that can be used to interact with the object.

Properties

A content item stored in the repository has an associated object to store its metadata. Since metadata is stored in object properties, the terms metadata and properties are used interchangeably. 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, user objects and permission set objects do not have any associated content. Users and permission sets are discussed in chapters related to security.

For More Information:
Each object property has a data type, which can be one of 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. Object IDs are discussed in detail later in this chapter.

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. For example, a document may have object_name='LoanApp_1234567891.txt' and keywords='John Doe','application','1234567891'.

The following figure shows a visual representation of this object. Typically, only properties are shown on the object while methods are shown when needed. Furthermore, only the properties relevant to the discussion are shown. Objects will be illustrated in this manner throughout the book:

![Object Diagram](image)

### Methods

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 r_lock_owner property with the name of the user performing the checkout. Methods are usually invoked using Documentum Foundation Classes (DFCs) programatically, though they can be indirectly invoked using API. In general, Documentum Query Language (DQL) cannot be used to invoke arbitrary methods on objects. DQL is discussed later in this chapter. DFC is discussed in *Chapter 4, Architecture* but DFC programming is beyond the scope of this book.
Note that the term method may be used in two different contexts within Documentum. A method as a defined operation on an object 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.

Working with objects
In Chapter 2, Working with Content we used Webtop for performing various operations on documents, where the term document referred to an object with content. Some of these operations are not specific to content and apply to objects in general. For example, checkout and checkin can be performed on contentless objects as well. On the other hand, import, export, and renditions deal specifically with content. Talking specifically about operations on metadata, we can view, modify, and export object properties using Webtop.

Viewing and editing properties
Using Webtop, object properties can be viewed using the View | Properties menu item, shortcut P, or the right-click context menu. The following screenshot shows the properties of the example object discussed earlier. Note that the same screen can be used to modify and save the properties as well.
Objects and Types

Multiple objects can be selected before viewing properties. In this case, a special dialog shows the common properties for the selected objects, as shown in the following figure. Any changes made on this dialog are applied to all the selected objects.

On the properties screen, single-valued properties can be edited directly while repeating properties provide a separate screen for editing through Edit links. Some properties cannot be modified by users at any time. Other properties may not be editable because object security prevents it or if the object is immutable. Security is discussed in other chapters while object immutability is discussed next.

Object immutability

Certain operations on an object mark it as immutable, which means that object properties cannot be changed. An object is marked immutable by setting r Immutable_flag to true. Content Server prevents changes to the content and metadata of an immutable object with the exception of a few special attributes that relate to the operations that are still allowed on immutable objects. For example, users can set a version label on the object, link the object to a folder, unlink it from a folder, delete it, change its lifecycle, and perform one of the lifecycle operations such as promote/demote/suspend/resume. The attributes affected by the allowed operations are allowed to be updated.
An object is marked immutable in the following situations:

- When an object is versioned or branched, it becomes an old version and is marked immutable.
- An object can be frozen which makes it immutable and imposes some other restrictions. Some virtual document operations can freeze the involved objects as discussed in Chapter 15, Virtual Documents.
- A retention policy can make the documents under its control immutable.

Certain operations such as unfreezing a document can reset the immutability flag making the object changeable again.

Exporting properties

Metadata can be exported from repository lists, such as folder contents and search results. Property values of the objects are exported and saved as a .csv (comma-separated values) file, which can be opened in Microsoft Excel or in a text editor. Metadata export can be performed using Tools | Export to CSV menu item or the right-click context menu. Before exporting the properties, the user is able to choose the properties to export from the available ones.

Object types

Objects in a repository may represent different kinds 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. Every time Content Server creates an object, it needs to determine the properties and methods that the object is going to possess. This information comes from an object type (also referred to as type).

The term attribute is synonymous with property and the two are used interchangeably. It is common to use the term attribute when talking about a property name and to use property when referring to its value.

We will use a dot notation to indicate that an attribute belongs to an object or a type. For example, objectA.title or dm_sysobject.object_name. This notation is succinct and unambiguous and is consistent with many programming languages.

An object type is a template for creating objects. In other words, an object is an instance of its type. A Documentum repository contains many predefined types and allows addition of new user-defined types (also known as custom types). User-defined types offer important capabilities and are described in detail in Chapter 11, Custom Types.
The most commonly used predefined object type for storing documents in the repository is `dm_document`. We have already seen how folders are used to organize documents. **Folders** are stored as objects of type `dm_folder`. A **cabinet** is a special kind of folder that does not have a parent folder and is stored as an object of type `dm_cabinet`. Users are represented as objects of type `dm_user` and a group of users is represented as an object of `dm_group`. Workflows use a process definition object of type `dm_process`, while the definition of a lifecycle is stored in an object of type `dm_policy`. These object types are described in more detail in later chapters. The following figure shows some of these types:
Just like everything else in the repository, a type is also represented as an object, which holds structural information about the type. This object is of type `dm_type` and stores information such as the name of the type, name of its supertype, and details about the attributes in the type. The following figure shows an object of type `dm_document` and an object of type `dm_type` representing `dm_document`. It also indicates how the type hierarchy information is stored in the object of type `dm_type`.

![Diagram showing type hierarchy](image)

The types present in the repository can be viewed using Documentum Administrator (DA). The following screenshot shows some attributes for the type `dm_sysobject`. This screen provides controls to scroll through the attributes when there are a large number of attributes present. The **Info** tab provides information about the type other than the attributes.

![Screen shot showing type properties](image)

For More Information:
Objects and Types

While the obvious use of a type is to define the structure and behavior of one kind of object, there is another very important utility of types. A type can be used to refer to all the objects of that type as a set. For example, queries restrict their scope by specifying a type where only the objects of that type are considered for matches. In our example scenario, the loan officer may want to search for all loan applications assigned to her. This query will be straightforward if there is an object type for loan applications. 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 the Chapter 5, User and Privileges.

Type names and property names

Each object type uses an internal type name, such as \texttt{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 \texttt{dm\_document} has the label Document.

Conventionally, internal names of predefined (defined by Documentum for Content Server or other client products) types start with \texttt{dm}, as described here:

- \texttt{dm\_}: (general) represents commonly used object types such as \texttt{dm\_document}, which is generally used for storing documents.
- \texttt{dmr\_}: (read only) represents read-only object types such as \texttt{dmr\_content}, which stores information about a content file.
- \texttt{dmi\_}: (internal) represents internal object types such as \texttt{dmi\_workitem}, which stores information about a task.
- \texttt{dmc\_}: (client) represents object types supporting Documentum client applications. For example, \texttt{dmc\_calendar} objects are used by Collaboration Services for holding calendar events.

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

- \texttt{r\_:} (read only) normally indicates that the property is controlled by the Content Server and cannot be modified by users or applications. For example, \texttt{r\_object\_id} represents the unique ID for the object. On the other hand, \texttt{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.

For More Information:

• \textit{i	extunderscore} (internal) is similar to \textit{r	extunderscore} except that this property is used internally by the Content Server and normally not seen by users and applications. As discussed in the previous chapter, \textit{i	extunderscore\_chronicle\_id} binds all the versions together in to a version tree and is managed by the Content Server.

• \textit{a	extunderscore} (application) 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 \textit{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 \textit{system} or non-application attributes, in general.

• \textit{\_} (computed) 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 \textit{\_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.

\section*{Type hierarchy}

It is common for different types to be related in some way and to share attributes and methods. In true object-oriented style, Documentum allows persistent types to be organized in an \textit{inheritance\textunderscore} based \textbf{type hierarchy}. A type can have none or one \textbf{supertype} and it inherits all the supertype attributes as its own. The complete set of attributes belonging to a type is the union of the inherited attributes and attributes explicitly defined for that type. In this relationship, the new type is called a \textbf{subtype}. A type with no supertype is called a \textbf{null type}. Null types are discussed in \textit{Chapter 11, Custom Types}.

The \textit{super} and \textit{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:

```
\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{type_hierarchy.png}
\caption{A type with no supertype is called a \textbf{null type}. Null types are discussed in \textit{Chapter 11, Custom Types}.}
\end{figure}
```

For More Information:

Objects and Types

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 such relationships are visually represented together, they create a tree structure 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:

```
persistent
object

dm_acl  dm_sysobject  dm_user

dm_document  dm_folder

 dm_cabinet
```

dm_document is an important type since it represents a document and is one of the most commonly used types. 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. Recall 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 and its subtypes which are not of the type dm_document.

**Object ID**

r_object_id is a special property of every persistent object. It is used to uniquely identify the object and encodes some information within the property itself. It is a 16-character string value where each character is a hex (hexadecimal) digit.

For More Information:

The first two digits constitute a **type 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. The next six digits represent the **repository ID** – a numeric identifier assigned to the repository. The last eight digits represent a **unique ID within the repository** and this ID is generated by the Content Server. The following figure illustrates the structure of the object ID and also shows the decimal values corresponding to the hex values.

```
<table>
<thead>
<tr>
<th>r_object_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>09</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>09</td>
</tr>
</tbody>
</table>
```

Note that EMC 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.

**Type categories**

Object types are categorized into **standard** and special categories for internal management by Content Server. A type is a standard type if it is not in one of the special categories. Most of the commonly used types are standard types.

The special object type categories are:

- **Shareable**: Shareable object types work in conjunction with lightweight object types. A single instance of a shareable type can be shared among many lightweight objects.
- **Lightweight**: Lightweight object types are used to minimize storage for multiple objects that share common system information. The shared properties reside in an instance of a shareable type and the rest of the properties reside in the lightweight objects. A lightweight type is a subtype of a shareable type.

For More Information:

Objects and Types

- **Aspect property:** Aspects enable addition of properties to any object regardless of its type. Aspect property object types are used internally for managing properties associated with aspects. Users and client applications are not aware of these types.

- **Data table:** Data table is a collaboration feature that enables users to manage structured collection of information.

Since shareable and lightweight object types are used together they are discussed together later in this chapter. Aspect property types are internal types not visible to the users so they are not discussed further. However, the corresponding feature for users – aspects, is discussed later in this chapter. Data table object types are used by the optional Collaboration Services and are not discussed further.

Content Server explicitly identifies the category of an object type using the `dm_type.type_category` attribute, which can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Standard object type</td>
</tr>
<tr>
<td>1</td>
<td>Aspect property object type</td>
</tr>
<tr>
<td>2</td>
<td>Shareable object type</td>
</tr>
<tr>
<td>4</td>
<td>Lightweight object type</td>
</tr>
<tr>
<td>8</td>
<td>Data table type</td>
</tr>
</tbody>
</table>

Object persistence

Objects 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 attributes:

- **r_object_id:** This is used for uniquely identifying the object and is assigned by Content Server. This property has been described earlier in this chapter.

- **i_vstamp:** This is used internally for managing object updates and holds the number of committed transactions that have altered this object.

- **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.

For More Information:

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. There is also an internal attribute named `i_position`, which defines the order of the values. 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 the table `dm_sysobject_r`.

While there is no requirement for two repeating attributes to be related there is no prohibition either. Indeed, various types have two or more repeating attributes that are related and correspond to one another by index position. For example, `dm_policy` represents a lifecycle and has several repeating attributes which correspond to each other by index position where each index represents one state. Among these attributes, `state_description[3]` will hold the description for the state named in `state_name[3]`, whose internal state number is stored in `i_state_no[3]`. Lifecycles are the subject of Chapter 13.

This storage scheme 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.
Objects and Types

Consider an example where a custom type `dq_document` has `dm_document` as its supertype, as shown in the following figure:

The following figure illustrates persistence for an object of `dq_document`. This figure only shows a small number of attributes for brevity. 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. However, all such persistence tables have the `r_object_id` column, which is used for joining information from other tables. Also note that `dm_document` does not add any properties of its own and `dq_document` does not add any repeating properties so the corresponding tables are absent from this figure. Another example of a type that doesn't use one of the tables is `dm_folder`, which has two repeating properties but no single-valued properties of its own.
Looking at the figure above, you may be wondering how we would know that this is an object of dq_document since the table name doesn't indicate that. In fact, these tables hold information for objects of all subtypes of dm_sysobject as well. The exact object type of an object is identified by the dm_sysobject_s.r_object_type column (not shown in the figure above).

It is useful to know how properties are stored in database tables though all the properties of an object can be queried together using DQL without any reference to these tables. Internally, Content Server uses database views that join appropriate tables to retrieve all the needed properties of the type together. Further, when multiple types are used in one DQL query the DQL parser applies appropriate table joins to achieve the intended effect.

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, a client config object holds the configuration parameters for sessions when a client attempts to connect to the Content Server.

### Content persistence

We have seen so far how metadata is persisted but it is not obvious how content is persisted and associated with its metadata. All sysobjects (objects of type dm_sysobject and its subtypes) other than folders (objects of type dm_folder and its subtypes) can have associated content. In Chapter 2, Working with Content we saw that a document can have content in the form of renditions as well as in primary format. How are these content files associated with a sysobject? In other words, how does Content Server know what metadata is associated with a content file? How does it know that one content file is a rendition of another one? Content Server manages content files using content objects, which (indirectly) point to the physical locations of content files and associate them with sysobjects.

### Locating content files

Recall that Documentum repositories can store content in various types of storage systems including a file system, a Relational Database Management System (RDBMS), a content-addressed storage (CAS), or external storage devices. Content Server decides to store each file in a location based on the configuration and the presence of products like Content Storage Services. In general, users are not concerned about where the file is stored since Content Server is able to retrieve the file from the location where it was stored. We will discuss the physical location of a content file without worrying about why Content Server chose to use that location.
Content object

Every content file in the repository has an associated content object, which stores information about the location of the file and identifies the sysobjects associated with it. These sysobjects are referred to as the parent objects of the content object.

A content object is an object of type dmr_content, whose key attributes are listed as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent_count</td>
<td>Number of parent objects.</td>
</tr>
<tr>
<td>parent_id</td>
<td>List of object IDs of the parent objects.</td>
</tr>
<tr>
<td>storage_id</td>
<td>Object ID of the store object representing the storage area holding the content.</td>
</tr>
<tr>
<td>data_ticket</td>
<td>A value used internally to retrieve the content. The value and its usage depend upon the type of storage used.</td>
</tr>
<tr>
<td>i_contents</td>
<td>When the content is stored in turbo storage, this property contains the actual content. If the content is larger than the size of this property (2000 characters for databases other than Sybase, 255 for Sybase), the content is stored in a dmi_subcontent object and this property is unused. If the content is stored in content-addressed storage, it contains the content address. If the content is stored in external storage, it contains the token used to retrieve the content.</td>
</tr>
<tr>
<td>rendition</td>
<td>Identifies if it's a rendition and its related behavior.</td>
</tr>
<tr>
<td></td>
<td>0 means original content.</td>
</tr>
<tr>
<td></td>
<td>1 means rendition generated by server.</td>
</tr>
<tr>
<td></td>
<td>2 means rendition generated by client.</td>
</tr>
<tr>
<td></td>
<td>3 means rendition not to be removed when its primary content is updated or removed.</td>
</tr>
<tr>
<td>format</td>
<td>Object ID of the format object representing the format of the content.</td>
</tr>
<tr>
<td>full_content_size</td>
<td>Content file size in bytes, except when the content is stored in external storage.</td>
</tr>
</tbody>
</table>
Object-content relationship

Content Server manages content objects while performing content-related operations. Content associated with a sysobject is categorized as **primary content** or a **rendition**. We have already learned about renditions in Chapter 2, Working with Content. A rendition is a content file associated with a sysobject that is not its primary content.

Content in the first content file added to a sysobject is called its primary content and its format is referred to as the **primary format** for the parent object. Any other content added to the parent object in the same format is also called primary content, though it is rarely done by users manually. This ability to add multiple primary content files is typically utilized programmatically by applications for their internal use.

While a sysobject can have multiple primary content files it is also possible for one content object to have multiple parent objects. This just means that a content file can be shared by multiple objects.

Putting it together

The details about content persistence can become confusing due to the number of objects involved and the relationships among various attributes. It becomes even more complicated when the full Content Server capabilities (such as multiple content files for one sysobject) are manifested. We will look at a simple scenario to visually grasp how content persistence works in common situations.

Documentum provides multiple options for locating the content file. DFC provides the `getPath()` method and DQL provides `get_file_url` administration method for this purpose. This section has been included to satisfy the reader's curiosity about content persistence and works through the information manually. This discussion can be treated as supplementary to technical fundamentals.

Let's revisit the renditions we saw in Chapter 2, Working with Content. The sysobject is named `paystub.jpg`. The primary content file is in `jpg` format and the rendition is in `pdf` format, as shown in the following figure:

![Rendition Example]

For More Information:

The following figure shows the objects involved in the content persistence for this document. The central object is of type `dm_document`. The figure also includes two content objects and one format object. Let's try to understand the relationships by asking specific questions.

How many content files, primary or renditions, are there for the document `paystub.jpg`? This question can be answered by looking for the corresponding content objects. We look for `dmr_content` objects that have the document's object ID in one of their `parent_id` values. This figure shows that there are two such content objects.

Which of these content objects represents the primary content and which one is a rendition? This can be determined by looking at the `rendition` attribute. The content object on the left shows `rendition=0`, which indicates primary content. The content object on the right shows `rendition=2`, which indicates rendition generated by client (recall that we manually imported this rendition).

What is the primary format for this document? This is easy to answer by looking at the `a_content_type` attribute on the document itself. If we need to know the format for a content object we can look for the `dm_format` object which has the same object ID as the value present in the `format` property of the content object. In the figure above, the format object for the primary content object is shown which represents a JPEG image. Thus, the format determined for the primary content of the object is expected to match the value of `a_content_type` property of the object. The format object for the rendition is not shown but it would be PDF.

What is the exact physical location of the primary content file? As mentioned in the beginning of this section, there are DFC and DQL methods which can provide this information. For understanding content persistence, we will deduce this manually for a file store, which represents storage on a file system. For other types of storage,
an exact location might not be evident since we need to rely on the storage interface to access the content file. Deducing the exact file path requires the ability to convert a decimal number to a hexadecimal (hex) number; this can be done with pen and paper or using one of the free tools available on the Web. Also remember that negative numbers are represented with what is known as a 2's-complement notation and many of these tools either don't handle 2's complement or don't support enough digits for our purposes.
There are two parts of the file path—the root path for the file store and the path of the file relative to this root path. In order to figure out the root path, we identify the file store first. Find the dm_filestore object whose object ID is the same as the value in storage_id property of the content object. Then find the dm_location object whose object name is the same as the root property on the file store object. The file_system_path property on this location object has the root path for the file store, which is C:\Documentum\data\localdev\content_storage_01 in the figure above.

In order to find the relative path of the content file, we look at data_ticket (data type integer) on the content object. Find the 8-digit hex representation for this number. Treat the hex number as a string and split the string with path separators (slashes, / or \ depending on the operating system) after every two characters. Suffix the right-most two characters with the file extension (.jpg), which can be inferred from the format associated with the content object. Prefix the path with an 8-digit hex representation of the repository ID. This gives us the relative path of the content file, which is 00000010\80\00\09\be.jpg in the figure above. Prefix this path with the file store root path identified earlier to get the full path of the content file.

Content persistence in Documentum appears to be complicated at first sight. There are a number of separate objects involved here and that is somewhat similar to having several tables in a relational database when we normalize the schema. At a high level, this complexity in the content persistence model serves to provide scalability, flexibility by supporting multiple kinds of content stores, and ease of managing changes in such an environment.

**Lightweight and shareable object types**

So far we have primarily dealt with standard types. Lightweight and shareable object types work together to provide performance improvements, which are significant when a large number of lightweight objects share information. The key performance benefits are in terms of savings in storage and in the time it takes to import a large number of documents that share metadata. These types are suitable for use in transactional and archival applications but are not recommended for traditional content management.

The term transactional content (as in business transactions) was coined by Forrester Research to describe content typically originating from external parties, such as customers and partners, and driving transactional back-office business processes. **Transactional Content Management (TCM)** unifies process, content, and compliance to support solutions involving transactional content. Our example scenario of mortgage loan approval process management is a perfect example of TCM. It involves numerous types of documents, several external parties, and sub-processes.
implementing parts of the overall process. Lightweight and shareable types play a central role in the **High Volume Server**, which enhances the performance of Content Server for TCM.

A **lightweight object type** (also known as **LwSO** for **Lightweight SysObject**) is a subtype of a **shareable type**. When a lightweight object is created, it references an object of its shareable supertype called the **parent object** of the lightweight object. Conversely, the lightweight object is called the **child object** of the shareable object. Additional lightweight objects of the same type can share the same parent object. These lightweight objects share the information present in the common parent object rather than each carrying a copy of that information.

In order to make the best use of lightweight objects we need to address a couple of questions. When should we use lightweight objects? Lightweight objects are useful when there are a large number of attribute values that are identical for a group of objects. This redundant information can be pushed into one parent object and shared by the lightweight objects.

What kind of information is suitable for sharing in the parent object? System-managed metadata, such as policies for security, retention, storage, and so on, are usually applied to a group of objects based on certain criteria. For example, all the documents in one loan application packet could use a single ACL and retention information, which could be placed into the shareable parent object. The specific information about each document would reside in a separate lightweight object. ACLs define permissions on objects and are discussed in Chapter 7, **Object Security**.

---

For More Information:  
Lightweight object persistence

Persistence for lightweight objects works much the same way it works for objects of standard types, with one exception. A lightweight object is a subtype of a shareable type and these types have their separate tables as usual. For a standard type, each object has separate records in all of these tables, with each record identified by the object ID of the object. However, when multiple lightweight objects share one parent object there is only one object ID (of the parent object) in the tables of the shareable type. The lightweight objects need to refer to the object ID of the parent object, which is different from the object ID of any of the lightweight objects, in order to access the shared properties. This reference is made via an attribute named `i_sharing_parent`, as shown in the last figure.

Materialization of lightweight objects

A group of lightweight objects sharing a parent object can be considered to be tethered to the parent object. They are forced to share the parent object's properties with the other lightweight objects and are said to be unmaterialized in this state. Suppose that one (or several) of the lightweight objects needs to change some of the shared properties while others need to retain the existing values. In this scenario, it is not possible to just change the property on the parent object. The lightweight object needs to break away from the rest and also needs to take a private copy of the properties of the parent object so that it can have different values for these properties.

When a lightweight object gets its own private copy of the parent object, it is said to be materialized. This terminology is similar to that used by RDBMS systems where materialized views may result in copies of previously shared column values. The private copy of the parent object uses the object ID of the materialized lightweight object. This state is similar to the standard type persistence, except that the join continues to use `i_sharing_parent` on the lightweight object table.

Materialization also results in a behavior change related to deletion. Deleting an unmaterialized lightweight object leaves the parent object alone, even if it was the last lightweight object pointing to the parent object. Deleting a materialized lightweight object always deletes its private copy of the parent object.

A lightweight object type definition includes specification of the materialization behavior, with the following options:

- Auto materialization—materialize automatically when certain operations occur
- Materialization on request—only materialize when explicitly requested
- Disallow materialization—do not materialize

For More Information:

Aspects

Aspects extend behavior and attributes for objects. An object type defines the properties and behavior of all objects of that type (and of its subtypes, via inheritance). In contrast, aspects enable us to attach additional behavior and attributes to an object, regardless of its type. Using aspects can speed up development and improve code reuse because they offer the possibility of extending attributes and behavior without altering the underlying type definitions.

In our example scenario, suppose that the mortgage lending company has a training program and they want to enhance it by providing examples of real-life situations and how their organization handles them. For this purpose, they want to tag documents that serve as good examples for handling specific scenarios. Irrespective of the type of document, they want to be able to add the following attributes to an example document:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>example_title</td>
<td>A single-valued string describing the example in a few words.</td>
</tr>
<tr>
<td>example_detail</td>
<td>A repeating string where each value describes a reason why this is an example.</td>
</tr>
</tbody>
</table>

In this scenario, we could create an aspect named Exemplary with these two attributes. We don’t need to modify any of the existing types nor do we have to create any new standard types. These attributes can be attached to any document and only to the documents that need it.

Restrictions

Even though aspects can be attached regardless of the type of object, aspects cannot be attached to arbitrary types of objects. Aspects can be attached under the following restrictions on types:

- An aspect can be attached to any sysobject
- An aspect can be attached to any object of null type (type with no supertype), once the type has been altered to allow aspects

It is also possible to attach multiple aspects to one object but one aspect cannot be attached multiple times to the same object.

If an aspect’s name contains a dot, such as jumbo.loan, it cannot have any properties. Why would we want an aspect without any properties? An aspect without any properties can serve as a flag where the presence of this flag can be checked on objects.
Default aspects

Even though aspects are not automatically tied to object types sometimes we may have a need to do just that (often for convenience). For example, we may have an existing type called dq_loan_doc and we may want all of its objects (and of its subtypes) to have an aspect dq_web_viewable attached. If an aspect is associated with an object type, the aspect is automatically attached to each new object of the specified object type. Such an aspect is called a default aspect for the specified type. Of course, this aspect can still be attached to objects of other types (subject to the restrictions mentioned earlier). Default aspects for an object type are recorded in dmi_type_info.default_aspects for the type.

What is the benefit of having a default aspect? If we need an aspect on every object of a type then making it a default aspect saves the effort of manually or programmatically attaching the aspect to each object. Why wouldn't we just alter the type to add those attributes to the type definition? Maybe the type is not allowed to be altered. Maybe it really needs to be an aspect and it is applicable to objects of other types as well.

Multiple default aspects can be associated with one object type. An object type inherits all the default aspects defined for its supertypes. When we add a default aspect to a type, the newly added aspect is only associated with new instances created after the addition of the default aspect. Existing instances of the type or its subtypes are not affected. Similarly, removal of a default aspect does not affect the existing instances of the type.

Content Server creates and manages an internal object type for each aspect. The names and definitions of aspect properties are stored in this internal type. The internal type is named dmi_type_id, where type_id is the object ID of the internal type.

Aspects provide an alternative to placing attributes on a standard object type. However, this option should be exercised carefully as a large number of aspects or ill-designed aspects can make usage and maintenance difficult. Remember that aspects are most useful when they are not needed on all the objects of a type but may be needed on objects of types that are unrelated via an inheritance hierarchy.

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 well-formed query in ANSI SQL is also a well-formed DQL query. However, successful execution of a well-formed
query depends on various factors such as existence of tables and availability of functions used in the query. DQL queries can be executed using IDQL (Interactive DQL shell), Documentum Administrator, Webtop, or programmatically through DFC applications. The results returned by a query are collectively referred to as a result set. When executing a query programmatically, the result set is represented by a collection object, which is a non-persistent object.

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 the SELECT, UPDATE, and DELETE 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:

```sql
SELECT r_object_id, r_creation_date
FROM dm_document
WHERE object_name = 'LoanApp_1234567890.txt'
```

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

- **SELECT clause (selected values list):** The selected values list specifies the properties to be retrieved.
- **FROM clause:** The FROM clause specifies the object types to be queried.
- **WHERE clause:** The WHERE clause is optional and specifies the conditions for matching the objects 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 in place of an object type in most DQL queries and its columns can be used as properties in a DQL query.

For More Information:
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`. All the subtypes of `dm_document` are also included in the scope of this query. Among these objects, we only want to look at current versions of objects that have 'LoanApp_1234567890.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. Selection of non-current versions is discussed later.

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. Of course, the results can be further narrowed by conditions and arguments that are part of the query.

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:

```sql
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:

```sql
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:

```sql
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 attributes are both \texttt{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 \texttt{d.} and \texttt{f.}, where \texttt{d} and \texttt{f} are \textit{aliases} (unrelated to the aliases in alias sets to be discussed in later chapters) for the types in the \texttt{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 \texttt{FROM} clause.

When one or more repeating attributes are included in the selected attributes, the results may be returned in two ways. If \texttt{r\_object\_id} is included in the selected attributes, the values of each repeating attribute are returned as one string containing comma-separated values. In this case, one row is returned per matching object. The following query and its result illustrate this behavior. There are two values in \texttt{r\_version\_label} and three in \texttt{keywords} but they are all returned in one row per object:

\begin{verbatim}
SELECT r\_object\_id, object\_name, r\_version\_label, keywords
FROM dm\_document
WHERE object\_name = 'LoanApp_1234567891.txt
\end{verbatim}

\begin{verbatim}
r\_object\_id | object\_name | r\_version\_label | keywords
--------------|---------------|-----------------|------------------
0900001080007932 | LoanApp_1234567891.txt | 3.0,CURRENT Application, 1234567891,Loan
\end{verbatim}

If \texttt{r\_object\_id} is not present in the selected attributes, only one value per row is returned even for repeating attributes. This means that multiple rows may be returned for one object. The following query and its result illustrate this behavior. There are two values in \texttt{r\_version\_label} and three in \texttt{keywords}. As a result, only one value per row but multiple rows per object are returned:

\begin{verbatim}
SELECT object\_name, i\_position, r\_version\_label, keywords
FROM dm\_document
WHERE object\_name = 'LoanApp_1234567891.txt'
ORDER BY i\_position DESC
\end{verbatim}

\begin{verbatim}
object\_name | i\_position | r\_version\_label | keywords
--------------|--------------|-----------------|------------------
LoanApp_1234567891.txt | -1 | 3.0 | Loan
LoanApp_1234567891.txt | -2 | CURRENT | Application
LoanApp_1234567891.txt | -3 | 1234567891
\end{verbatim}

Note the use of \texttt{ORDER BY} clause and \texttt{i\_position} attribute for sorting the results. \texttt{DESC} indicates descending order. Multiple comma-separated attributes can be specified in the \texttt{ORDER BY} clause for sorting.
**Objects and Types**

It is possible to use an asterisk (*) instead of a list of attribute names to be returned. In general, the * returns a set of predefined attributes but the exact behavior depends on the queried types and the presence of certain hints in the query. For full details about the behavior of *, see the *DQL Reference Manual*.

A DQL query only works on the current versions of objects unless *(ALL)* is used after the type name. For example, the following query retrieves name and title for all versions:

```sql
SELECT object_name, title
FROM dm_document (ALL)
```

It is rare to run a select query without a *WHERE* clause because it doesn't filter objects of the specified type(s). A query without a *WHERE* clause may take an inordinately long time to return the results. The *WHERE* clause provides conditions or search criteria for narrowing down the search scope to find relevant objects.

**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. The following examples illustrate the usage of the *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 r_object_id, title, subject
FROM dm_document
WHERE object_name = 'LoanApp_1234567890.txt'
```

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 = '0900001080002514'
```

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

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

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 09/09/2009:

```sql
SELECT object_name
FROM dm_document
WHERE r_modify_date < DATE('09/09/2009')
```

In addition to a comparison with exact values, pattern matching can be performed against string attributes. The character % represents zero or more characters while _ represents exactly one character. Pattern matching is performed using LIKE and NOT LIKE predicates as shown in the following example.

```sql
SELECT r_object_id, title, subject
FROM dm_document
WHERE object_name LIKE 'LoanApp%'
```

This query matches all current documents whose names start with LoanApp. The use of NOT LIKE would match all current objects that LIKE would not have matched.

Pattern matching is great but how would we find names containing %rate, for example? Since % has a special meaning for the LIKE predicate, it needs to be escaped. Essentially, we need to distinguish the literal % from the pattern matching %. The following query illustrates how to escape a pattern-matching character.

```sql
SELECT r_object_id, title, subject
FROM dm_document
WHERE object_name LIKE '%$rate%' ESCAPE '\'
```

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:

```sql
UPDATE <type_name> OBJECT
<property_updates>
WHERE <condition>
```
Objects and Types

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 not be omitted as far as possible because omitting it would lead to all objects of the type being updated (that are not otherwise restricted because of lack of permissions, immutability, or being checked out by another user).

<type_name> is the type or an ancestor type of the object(s) to be updated. Recall from the tree terminology discussed in Chapter 2, Working with Content that an ancestor can be the parent or an ancestor's parent in a tree. Therefore, an ancestor type could be the supertype, supertype's supertype, and so on. Sometimes, the ancestor type is loosely referred to as supertype to include all these possibilities.

<property_updates> specify the property names and the corresponding values to be set.

The following example illustrates these concepts:

```
UPDATE dm_document OBJECT
SET title = 'John''s Loan Application',
SET subject = '1234567890'
WHERE r_object_id = '0900001080002514'
```

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 a registered table. <property_updates> usually takes the form SET <property_name> = <value>. If multiple properties are being updated, each SET clause is separated using a comma.

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.

In addition to setting literal (constant) values, it is possible to copy a value from one attribute to another, as shown here. This query copies the value of object_name into title:

```
UPDATE dm_document OBJECT
SET subject = object_name
WHERE r_object_id = '0900001080002514'
```

The following query updates or sets repeating attributes:

```
UPDATE dm_document OBJECT
SET authors[0] = 'John',
SET authors[1] = 'Jane'
WHERE r_object_id = '0900001080002514'
```
Note that if a repeating property, like authors in this example, needs to be updated, an 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. However, if we just need to append values at the end of the list we can use this format – `APPEND <property_name> = <value>`. Multiple values can be appended by using multiple `APPENDs` within the same update query.

Repeating values can be removed using `TRUNCATE`, as shown in the following example:

```sql
UPDATE dm_document OBJECT
TRUNCATE authors
WHERE r_object_id = '0900001080002514'
```

This query removes all values for authors. In another variation, `TRUNCATE authors [2]` will remove all values at index 2 and higher.

### DELETE query

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

```sql
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.

![Caution: Careful with DELETE queries.](image)

Let's look at an example of the `DELETE` query:

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

This query deletes all current 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 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.

For More Information:

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

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 current 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`, `/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 for all folder paths listed.

### 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. API is no longer officially supported since most API methods have been mapped to DFC methods.

### Documentum product notes

Normally, there is one content object per content file in the repository. However, if **Content Storage Services** are deployed and content duplication checking and prevention is enabled, multiple content objects may become associated with one content file. This happens when identical copies of a content file are found and removed.

For More Information:

Since objects, content objects, and content files can have many-to-many relationships; Content Server uses certain attributes on content objects to track these relationships. In order to uniquely identify these relationships, each content file is assigned a page number per parent object and the page number identifies this content file uniquely among the content files associated with that sysobject. Page numbers are useful for identifying the primary content that is the source of a rendition.

Typically, a rendition is associated with a parent object via a content object. However, renditions created by the media server can be alternatively connected to their source via a relation object.

**High Volume Server** provides transactional capabilities (as in business transactions) and features for rapid ingestion, efficient database storage, and reliable access to content. It can be used as a standalone repository for store-and-retrieval applications or as a transaction processing accelerator when coupled with a Documentum Content Server.

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 holding temporary objects. Aspects are a relatively new feature and its support in the product suite is still maturing.

**Learn more**

The topics discussed in this chapter can be further explored using the following resources:

- EMC Documentum Content Server 6.5 Fundamentals
- EMC Documentum Webtop 6.5 SP2 User Guide
- EMC Documentum System 6.5 Object Reference Manual
- EMC Documentum System 6.5 SP2 DQL Reference Manual
- EMC Documentum Retention Policy Services Administrator 6.5 SP1 User Guide
- EMC Documentum Transactional Content Management—A Detailed Review
- DFC Javadocs
- Two's complement—http://en.wikipedia.org/wiki/Two%27s_complement

For More Information:
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 \texttt{r\_object\_id} attribute?
3. What are the object type categories?
4. What is a type hierarchy? How are objects of standard types persisted in the repository database?
5. What are lightweight and shareable types? How are objects of these types persisted?
6. What are aspects? When should they be used?
7. What are the various ways of querying objects in a repository? What are some common DQL queries?
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.