What's New in DAO?

DAO 3.6

DAO 3.6 has been updated to use the Microsoft® Jet 4.0 database engine. This includes enabling all interfaces for Unicode. Data is now provided in unicode (internationally enabled) format rather than ANSI. No other new features were implemented.

Support for the dBase and Paradox ISAMs now require the installation of the Borland database engine. Additionally, Jet 4.0 does not offer a Visual FoxPro ISAM, you need to use the Visual FoxPro ODBC Driver instead. These changes do not affect 3.5x applications and only affect developers who explicitly update their applications to use DAO 3.6, because DAO 3.5x and DAO 3.6 can coexist on the same computer.

DAO 3.5

DAO 3.5 introduced a new client/server connection mode, called "ODBCDirect." ODBCDirect establishes a connection directly to an ODBC data source, without loading the Microsoft Jet database engine into memory, and is useful in situations where specific features of ODBC are required.

For Microsoft Jet databases, there are also new interfaces to expose Microsoft Jet's new partial replication feature.

Note  You can send DAO queries to a variety of different database servers with ODBCDirect, and different servers will recognize slightly different dialects of SQL. Therefore, context-sensitive Help is no longer provided for Microsoft Jet
SQL, although online Help for Microsoft Jet SQL is still included through the Help menu. Be sure to check the appropriate reference documentation for the SQL dialect of your database server when using either ODBCDirect connections or pass-through queries in Microsoft Jet-connected client/server applications.

New DAO 3.5 Interfaces for ODBCDirect

- **Connection** object — A connection to an **ODBC** database.

- **Cancel** method (on **Connection**, **QueryDef**, and **Recordset** objects) — Cancels execution of an asynchronous operation.

- **NextRecordset** method (on **Recordset** objects) — Retrieves the next set of records, if any, returned by a query that returned multiple sets of records in an **OpenRecordset** call, and indicates whether it successfully retrieved another set of records.

- **OpenConnection** method (on **Workspace** objects) — Opens a **Connection** object on an **ODBC data source**.

- **BatchCollisionCount** property (on **Recordset** objects) — Returns the number of records that did not complete during the last **batch update**.

- **BatchCollisions** property (on **Recordset** objects) — Returns an array of bookmarks indicating the rows that generated collisions in the last batch update.

- **BatchSize** property (on **Recordset** objects) — Sets or returns the number of statements sent back to the server in each batch.

- **Connection** property (on **Database** and **Recordset** objects) — Returns the **Connection** object that corresponds to the **Database**, or that owns the **Recordset**.

- **Database** property (on **Connection** objects) — Returns the name of the **Database** object that corresponds to the **Connection**.

- **DefaultCursorDriver** property (on **Workspace** objects) — Sets or returns the type of cursor driver used for ODBCDirect **Recordset** objects.
• **DefaultType** property (on **DBEngine** object) — Indicates what type of workspace (Microsoft Jet or **ODBCDirect**) will be created by the next **CreateWorkspace** method call.

• **Direction** property (on **Parameter** objects) — Indicates whether a **Parameter** object represents an input parameter, an output parameter, or both, or if the parameter is the return value from a stored procedure.

• **MaxRecords** property (on **QueryDef** objects) — Sets or returns the maximum number of records to return from a query.

• **OriginalValue** property (on **Field** objects) — Returns the value of a **Field** in the database that existed when the last batch update began.

• **Prepare** property (on **QueryDef** objects) — Returns a value that indicates whether the query should be prepared on the server as a temporary stored procedure with the ODBC **SQLPrepare** function prior to execution, or just executed using the ODBC **SQLExecDirect** function.

• **RecordStatus** property (on **Recordset** objects) — Returns a value that indicates the update status of the current record if it is part of a batch update.

• **StillExecuting** property (on **Connection**, **QueryDef**, and **Recordset** objects) — Returns a value indicating whether or not an asynchronous operation has finished executing.

• **UpdateOptions** property (on **Recordset** objects) — Returns a value that indicates how the WHERE clause is constructed for each record during a batch update, and how the update should be executed.

• **VisibleValue** property (on **Recordset** objects) — Returns a value currently in the database that is newer than the **OriginalValue** property as determined by a batch update conflict.

**New Capabilities with ODBCDirect**

**Server Connections**
Available only in the ODBCDirect object model, the new **Connection** object contains information about a connection to an ODBC data source, such as the server name, the data source name, and so on. It is similar to a **Database** object, and will look very familiar if you’ve ever opened a **Database** object on an ODBC data source. In fact, a **Connection** object and a **Database** object represent different references to the same object, and new properties on each of these two object types allow you to obtain a reference to the other corresponding object, which simplifies the task of converting existing ODBC client applications that use Microsoft Jet to use ODBCDirect instead.

**Batch Updates**

A new batch update cursor is available for client applications that need to work with a cursor without holding locks on the server or issue update requests one record at a time. Instead, the client stores update information on many records in a local buffer (or "batch"), and then issues a batch update.

Because of the time lag between opening a **Recordset** and sending a batch of updates from that **Recordset** back to the server, other users have an opportunity to change the original data before your changes are sent to the server, so your changes "collide" with another user's changes. Several new features are available to help you determine where such collisions have occurred, following a batch update, and give you some options for resolving them.

**Asynchronous Method Execution**

The **Execute**, **MoveLast**, **OpenConnection**, and **OpenRecordset** methods feature the **dbRunAsync** option. This allows the client application to do other tasks (such as loading forms, and so on) while the method is executing. You can also poll to see whether the task is complete, and terminate an asynchronous task.

**Client Support for ODBC Cursors**

Four different **Recordset** types support the following ODBC cursor types:

<table>
<thead>
<tr>
<th>ODBC Cursor</th>
<th>Recordset type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td><strong>dbOpenDynamic</strong> (New in DAO 3.5)</td>
</tr>
</tbody>
</table>
New DAO 3.5 Interfaces for the Microsoft Jet Database Engine

- **PopulatePartial** method (on Database objects) — Synchronizes any changes in a partial replica with the full replica, clears all records in the partial replica, and then repopulates the partial replica based on the current replica filters.

- **SetOption** method (on DBEngine object) — Overrides the registry values for the Microsoft Jet database engine for the duration of the current instance of DAO.

- **FieldSize** property (on Field objects) — Replaces the FieldSize method. Syntactically, their usage is the same, so this will not require changes to your existing code.

- **MaxRecords** property (on QueryDef objects) — Sets or returns the maximum number of records to return from a query.

- **ReplicaFilter** property (on TableDef objects) — Returns a value that indicates which subset of records is replicated to that table from a full replica.

- **PartialReplica** property (on Relation objects) — Indicates which Relation object should be considered when populating a partial replica from a full replica.

New Capabilities with the Microsoft Jet Database Engine

Partial Replication

Version 3.5 of the Microsoft Jet database engine allows users to replicate portions of a table instead of the whole table (only row restrictions are permitted,
not columns). There are two types of filters used in a partial replica — Boolean and relationship. Boolean filters select only rows that meet a certain criteria to limit the rows in a table that are replicated. DAO represents this filter with the ReplicaFilter property on a TableDef. Relationship filters enforce a relationship between partially replicated tables to limit the rows in a table that are replicated. With DAO, you can set the PartialReplica property on a Relation which allows that Relation to be used in partial replication.

**New Recordset Type**

In DAO 3.5, dbOpenForwardOnly is a new type argument for the OpenRecordset method. This new Recordset type behaves in the same way as a DAO 3.0 snapshot-type Recordset opened with the dbForwardOnly option.

**Run-time Registry Override**

The new SetOption method allows you to override Microsoft Jet Registry settings at run time. This lets you fine tune Microsoft Jet query performance, timeout delays, and so on.
DAO Overview

DAO enables you to use a programming language to access and manipulate data in local or remote databases, and to manage databases, their objects, and their structure.
Object Models

DAO supports two different database environments, or "workspaces."

- **Microsoft Jet workspaces** allow you to access data in Microsoft Jet databases, Microsoft Jet-connected ODBC data sources, and installable ISAM data sources in other formats, such as Paradox or Lotus 1-2-3.

- **ODBCDirect workspaces** allow you to access database servers through ODBC, without loading the Microsoft Jet database engine.

Use the Microsoft Jet workspace when you open a Microsoft Jet database (.mdb file) or other desktop ISAM database, or when you need to take advantage of Microsoft Jet's unique features, such as the ability to join data from different database formats.

The **ODBCDirect** workspace provides an alternative when you only need to execute queries or stored procedures against a back-end server, such as Microsoft SQL Server, or when your client application needs the specific capabilities of ODBC, such as batch updates or asynchronous query execution.

DAO Objects

There are 17 different DAO object types. You can declare new DAO object variables for any of the object types.

For example, the following Visual Basic for Applications (VBA) code creates object variables for a **Database** object, a dynaset-type **Recordset** object, and a **Field** object:

```vba
Dim dbsExample As Database
Dim rstExample As Recordset
Dim fldExample As Field

Set dbsExample = OpenDatabase("Biblio.mdb")
Set rstExample = dbsExample.OpenRecordset("Authors", dbOpenDyn)
Set fldExample = rstExample.Fields("Au_ID")
```
DAO Collections

Each DAO object type other than DBEngine also has a corresponding collection. A collection includes all the existing objects of that type. For example, the Recordsets collection contains all open Recordset objects. Each collection is "owned" by another object at the next higher level in the hierarchy. A Database object "owns" a Recordsets collection. Except for the Connection and Error objects, every DAO object has a Properties collection.

Most DAO objects have default collections and default properties. For example, the default collection of a Recordset object is the Fields collection and the default property of a Field object is the Value property. You can simplify your code by taking advantage of these defaults. For example, the following code sets the value of the PubID field in the current record:

rstExample!PubID=99

DBENGINE AND WORKSPACE OBJECTS

All DAO objects are derived from the DBEngine object. You can set the DefaultType property on the DBEngine object to determine the workspace type (Microsoft Jet or ODBCDirect) to create on subsequent CreateWorkspace method calls, or you can override this property with the type argument in the CreateWorkspace method itself. When your application creates a workspace, the appropriate library — the Microsoft Jet database engine or ODBC — is loaded into memory at that time.

You can open additional Workspace objects as needed. Each Workspace object has a user ID and password associated with it.
Using the Microsoft Jet Workspace

Opening a Database

To open a database, you simply open an existing Database object, or create a new one. This object can represent a Microsoft Jet database (.mdb file), an ISAM database (for example, Paradox), or an ODBC database connected through the Microsoft Jet database engine (also known as a "Microsoft Jet-connected ODBC database").

Data-Definition Language

You can use object variables and other DDL features to modify your database structure. For example, you can add a new Field object to an existing table with the following code:

Dim dbs As Database, tdf As TableDef, fld As Field
' Open a database.
Set dbs = OpenDatabase("Biblio.mdb")
' Open a TableDef.
Set tdf = dbs.TableDefs("Authors")
' Create a new field.
Set fld = tdf.CreateField("Address", dbText, 20)
' Append field to the TableDef Fields collection.
tdf.Fields.Append fld

This code creates a new object variable for a Field object and adds it to a TableDef object with the Append method. Because a TableDef object contains the definition of a table, the table now has a field named Address for entering data. In much the same way, you can create new tables and new indexes.

Data Manipulation

DAO provides an excellent set of data manipulation tools. You can create a Recordset object to conveniently query a database and manipulate the resulting set of records. The OpenRecordset method accepts an SQL string, or a QueryDef (stored query) name as a data source argument, or it can be opened from a QueryDef object or a TableDef object, using that object as its data source. The resulting Recordset object features an extremely rich set of
properties and methods with which to browse and modify data.

The **Recordset** object is available in four different types — Table, Dynaset, Forward-Only, and Snapshot.

**Transactions**

All **Database** objects opened against a **Workspace** object share a common transaction scope. That is, when you use the **BeginTrans** method on a **Workspace** object, it applies to all open databases within that **Workspace** object. In the same way, when you use the **CommitTrans** method against the **Workspace**, it applies to all open databases in the **Workspace** object.

**Replication**

You can use [database replication](https://msdn.microsoft.com/en-us/library/office/ff833872) to create and maintain **replicas** of a master Microsoft Jet database, using the **Synchronize** method to periodically update all or part of the replicas, or to copy new data from one replica to another. You can also restrict the update to only selected records, using the **ReplicaFilter** property, and then synchronize those records with the **PopulatePartial** method.

**Security**

You can restrict access to one or more .mdb databases or their tables using security settings established and managed by the Microsoft Jet database engine. In your code, you can establish **Group** and **User** objects to define the scope and level of **permissions** available to individual users on an object-by-object basis. For example, you can establish permissions for a specific user to provide read-only access to one table and full access to another.
Using the ODBC Direct Object Model

Connecting to a Database

A Connection object is similar to a Database object. In fact, a Connection object and a Database object represent different references to the same object, and properties on each of these two object types allow you to obtain a reference to the other corresponding object, which simplifies the task of converting ODBC client applications that use Microsoft Jet to use ODBC Direct instead. Use the OpenConnection method to connect to an ODBC data source. The resulting Connection object contains information about the connection, such as the server name, the data source name, and so on.

Queries

Although DAO does not support stored queries in an ODBC Direct workspace, a compiled query can be created as a QueryDef object and used to execute action queries, and can also be used to execute stored procedures on the server. The Prepare property lets you decide whether to create a private, temporary stored procedure on the server from a QueryDef before actually executing the query.

Parameter queries can also be passed to the server, using Parameter objects on the QueryDef. The Direction property lets you specify a Parameter as input, output, or both, or to accept a return value from a stored procedure.

Data Manipulation

Creating a Recordset object is a convenient way to query a database and manipulate the resulting set of records. The OpenRecordset method accepts an SQL string, or a QueryDef object (stored query) as a data source argument. The resulting Recordset object features an extremely rich set of properties and methods with which to browse and modify data.

The Recordset object is available in four different types — Dynamic, Dynaset, Forward-Only, and Snapshot — corresponding to ODBC cursor types — Dynamic, Keyset, Forward-only, and Static.
A batch update cursor library is available for client applications that need to work with a cursor without holding locks on the server or without issuing update requests one record at a time. Instead, the client stores update information on many records in a local buffer (or "batch"), and then issues a batch update.

**Asynchronous Method Execution**

The **Execute**, **MoveLast**, **OpenConnection**, and **OpenRecordset** methods feature the **dbRunAsync** option. This allows your client application to do other tasks (such as loading forms, for example) while the method is executing. You can check the **StillExecuting** property to see whether the task is complete, and terminate an asynchronous task with the **Cancel** method.
DAO provides built-in constants that you can use with methods or properties. These constants all begin with the letters db and are documented with the method or property to which they apply.

Legend:

- ⚖️ Read-only
- ⚖️.Read/write

**AllPermissions Property Constants (All Are ⚖️)****

For any Container or Document object:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecReadDef</td>
<td>Allows user to read the table definition, including column and index information.</td>
</tr>
<tr>
<td>dbSecWriteDef</td>
<td>Allows user to modify or delete the table definition, including column and index information.</td>
</tr>
</tbody>
</table>
index information.

- **dbSecRetrieveData**
  - Allows user to retrieve data from the Document object.

- **dbSecInsertData**
  - Allows user to add records.

- **dbSecReplaceData**
  - Allows user to modify records.

- **dbSecDeleteData**
  - Allows user to delete records.

The **Databases** container or any Document object in a Documents collection may include the following:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecDeleteData</td>
<td>Allows user to delete records.</td>
</tr>
<tr>
<td>dbSecDBAdmin</td>
<td>Allows user to replicate the database and change the</td>
</tr>
<tr>
<td></td>
<td>database password.</td>
</tr>
<tr>
<td>dbSecDBCreate</td>
<td>Allows user to create new databases.</td>
</tr>
<tr>
<td></td>
<td>This setting is valid only on the Databases container</td>
</tr>
<tr>
<td></td>
<td>in the workgroup information file (System.mdw).</td>
</tr>
<tr>
<td>dbSecDBExclusive</td>
<td>Allows user exclusive access to the database.</td>
</tr>
<tr>
<td>dbSecDBOpen</td>
<td>Allows user to open the database.</td>
</tr>
</tbody>
</table>

**Attributes Property Constants**

For any Field object, the Attributes property may include the following:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbFixedField</td>
<td>✨ Fixed field size (default for Numeric fields)</td>
</tr>
<tr>
<td>dbVariableField</td>
<td>✨ Variable field size (Text fields only)</td>
</tr>
<tr>
<td>dbAutoIncrField</td>
<td>✨ New record field value incremented to unique Long integer (in a Microsoft</td>
</tr>
<tr>
<td></td>
<td>Jet workspace, available only on TableDef objects opened from .mdb files)</td>
</tr>
<tr>
<td>dbUpdatableField</td>
<td>✨ Field is updatable</td>
</tr>
</tbody>
</table>
dbDescending

Field sorted in descending order (Microsoft Jet workspaces only)

dbHyperlinkField

The field contains hyperlink information (Memo fields in Microsoft Jet workspaces only)

dbSystemField

The field is a replication field (on a TableDef object in Microsoft Jet databases only)

For any Relation object, the Attributes property may include the following:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbRelationUnique</td>
<td>One-to-one relationship</td>
</tr>
<tr>
<td>dbRelationDontEnforce</td>
<td>Relationship not enforced (no referential integrity)</td>
</tr>
<tr>
<td>dbRelationInherited</td>
<td>Relationship exists in the database containing the two linked tables</td>
</tr>
<tr>
<td>dbRelationUpdateCascade</td>
<td>Updates cascade</td>
</tr>
<tr>
<td>dbRelationDeleteCascade</td>
<td>Deletions cascade</td>
</tr>
<tr>
<td>dbRelationLeft</td>
<td>Microsoft Access only. In Design view, display a LEFT JOIN as the default join type.</td>
</tr>
<tr>
<td>dbRelationRight</td>
<td>Microsoft Access only. In Design view, display a RIGHT JOIN as the default join type.</td>
</tr>
</tbody>
</table>

For any TableDef object, the Attributes property may include the following:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbAttachExclusive</td>
<td>Opens a linked Microsoft Jet database engine table for exclusive use.</td>
</tr>
<tr>
<td>dbAttachSavePWD</td>
<td>Saves user ID and password for linked remote table.</td>
</tr>
<tr>
<td>dbSystemObject</td>
<td>System table</td>
</tr>
</tbody>
</table>
### CollatingOrder Property Constants (All Are 📅)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSortArabic</td>
<td>Arabic collating order</td>
</tr>
<tr>
<td>dbSortChineseSimplified</td>
<td>Simplified Chinese collating order</td>
</tr>
<tr>
<td>dbSortChineseTraditional</td>
<td>Traditional Chinese collating order</td>
</tr>
<tr>
<td>dbSortCyrillic</td>
<td>Russian collating order</td>
</tr>
<tr>
<td>dbSortCzech</td>
<td>Czech collating order</td>
</tr>
<tr>
<td>dbSortDutch</td>
<td>Dutch collating order</td>
</tr>
<tr>
<td>dbSortGeneral</td>
<td>English, German, French, and Portuguese collating order</td>
</tr>
<tr>
<td>dbSortGreek</td>
<td>Greek collating order</td>
</tr>
<tr>
<td>dbSortHebrew</td>
<td>Hebrew collating order</td>
</tr>
<tr>
<td>dbSortHungarian</td>
<td>Hungarian collating order</td>
</tr>
<tr>
<td>dbSortIcelandic</td>
<td>Icelandic collating order</td>
</tr>
<tr>
<td>dbSortJapanese</td>
<td>Japanese collating order</td>
</tr>
<tr>
<td>dbSortKorean</td>
<td>Korean collating order</td>
</tr>
<tr>
<td>dbSortNeutral</td>
<td>Neutral collating order</td>
</tr>
<tr>
<td>dbSortNorw</td>
<td>Norwegian and Danish collating order</td>
</tr>
<tr>
<td>dbSortPDXIntl</td>
<td>Paradox international collating order</td>
</tr>
<tr>
<td>dbSortPDXNor</td>
<td>Paradox Norwegian and Danish collating order</td>
</tr>
<tr>
<td>dbSortPDXSwe</td>
<td>Paradox Swedish and Finnish collating order</td>
</tr>
<tr>
<td>dbSortPolish</td>
<td>Polish collating order</td>
</tr>
<tr>
<td>dbSortSlovenian</td>
<td>Slovenian collating order</td>
</tr>
<tr>
<td>dbSortSpanish</td>
<td>Spanish collating order</td>
</tr>
<tr>
<td>dbSortSwedFin</td>
<td>Swedish and Finnish collating order</td>
</tr>
<tr>
<td>dbSortThai</td>
<td>Thai collating order</td>
</tr>
</tbody>
</table>
dbSortTurkish  Turkish collating order
dbSortUndefined Collating order undefined or unknown

**DefaultCursorDriver** Property (All are \(\text{accessible}\))

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbUseDefaultCursor</td>
<td>(Default) Uses <a href="#">server-side cursors</a> if the server supports them; otherwise uses the ODBC Cursor Library.</td>
</tr>
<tr>
<td></td>
<td>Always uses the ODBC Cursor Library.</td>
</tr>
<tr>
<td>dbUseODBCCursor</td>
<td>This option provides better performance for small result sets, but degrades quickly for larger result sets.</td>
</tr>
<tr>
<td></td>
<td>Always uses <a href="#">server-side cursors</a>. For most large operations this option provides better performance, but might cause more network traffic.</td>
</tr>
<tr>
<td>dbUseServerCursor</td>
<td>Always uses the FoxPro Cursor Library. This option is required for performing batch updates.</td>
</tr>
<tr>
<td></td>
<td>Opens all cursors (that is, <a href="#">Recordset</a> objects) as forward-only type, read-only, with a rowset size of 1. Also known as &quot;cursorless queries.&quot;</td>
</tr>
<tr>
<td>dbUseClientBatchCursor</td>
<td></td>
</tr>
<tr>
<td>dbUseNoCursor</td>
<td></td>
</tr>
</tbody>
</table>

**Direction** Property Constants (All Are \(\text{accessible}\))

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbParamInput</td>
<td>(Default) Passes information to the procedure.</td>
</tr>
<tr>
<td></td>
<td>Passes information both to and from the procedure.</td>
</tr>
<tr>
<td>dbParamInputOutput</td>
<td>Returns information from the procedure as in an output parameter in SQL.</td>
</tr>
<tr>
<td>dbParamOutput</td>
<td>Passes the return value from a</td>
</tr>
</tbody>
</table>
dbParamReturnValue procedure.

**EditMode Property** Constants (All Are 🄪)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbEditNone</td>
<td>No editing operation in effect.</td>
</tr>
<tr>
<td>dbEditInProgress</td>
<td>Edit method invoked.</td>
</tr>
<tr>
<td>dbEditAdd</td>
<td>AddNew method invoked.</td>
</tr>
</tbody>
</table>

**Permissions Property** Constants (All are 🄪)

For any Container object, the Permissions property may include the following:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecNoAccess</td>
<td>Denies user access to the object.</td>
</tr>
<tr>
<td>dbSecFullAccess</td>
<td>Allows user full access to the object.</td>
</tr>
<tr>
<td>dbSecDelete</td>
<td>Allows user to delete the object.</td>
</tr>
<tr>
<td>dbSecReadSec</td>
<td>Allows user to read the object's security-related information.</td>
</tr>
<tr>
<td>dbSecWriteSec</td>
<td>Allows user to alter access permissions.</td>
</tr>
<tr>
<td>dbSecWriteOwner</td>
<td>Allows user to change the Owner property setting.</td>
</tr>
</tbody>
</table>

For any database Container, the Permissions property may include any of the following (All are 🄪):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecDBAdmin</td>
<td>Gives user permission to make a database replicable and change the database password.</td>
</tr>
<tr>
<td>dbSecDBCreate</td>
<td>Allows user to create new databases (valid only on the databases Container object in the system database).</td>
</tr>
<tr>
<td>Constant</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dbSecCreate</td>
<td>Allows user to create new tables (valid only with a Container object that represents a table).</td>
</tr>
<tr>
<td>dbSecReadDef</td>
<td>Allows user to read the table definition, including column and index information.</td>
</tr>
<tr>
<td>dbSecWriteDef</td>
<td>Allows user to modify or delete the table definition, including column and index information.</td>
</tr>
<tr>
<td>dbSecRetrieveData</td>
<td>Allows user to retrieve data from the document.</td>
</tr>
<tr>
<td>dbSecInsertData</td>
<td>Allows user to add records.</td>
</tr>
<tr>
<td>dbSecReplaceData</td>
<td>Allows user to modify records.</td>
</tr>
<tr>
<td>dbSecDeleteData</td>
<td>Allows user to delete records.</td>
</tr>
</tbody>
</table>

For any Document object, the Permissions property may include any of the following (All are $\mathcal{C}$):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecCreate</td>
<td>Allows user to create new tables (valid only with a Container object that represents a table).</td>
</tr>
<tr>
<td>dbSecDBCreate</td>
<td>Allows user to create new databases (valid only on the databases Container object in the system database).</td>
</tr>
<tr>
<td>dbSecDBOpen</td>
<td>Allows user to open the database.</td>
</tr>
</tbody>
</table>
**dbSecDBExclusive**  Allows user exclusive access.
**dbSecDelete**  Allows user to delete the object.
**dbSecDeleteData**  Allows user to delete records.
**dbSecFullAccess**  Allows user full access to the object.
**dbSecInsertData**  Allows user to add records.
**dbSecReadDef**  Allows user to read the table definition, including column and index information.
**dbSecReadSec**  Allows user to read the object's security-related information.
**dbSecReplaceData**  Allows user to modify records.
**dbSecRetrieveData**  Allows user to retrieve data from the document.
**dbSecWriteDef**  Allows user to modify or delete the table definition, including column and index information.
**dbSecWriteSec**  Allows user to alter access permissions.
**dbSecWriteOwner**  Allows user to change the **Owner** property setting.

**Prepare** Property Constants (All Are [ ])

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dbQPrepare</strong></td>
<td>(Default) The statement is prepared (that is, the ODBC <strong>SQLPrepare</strong> API is called).</td>
</tr>
<tr>
<td><strong>dbQUnprepare</strong></td>
<td>The statement is not prepared (that is, the ODBC <strong>SQLExecDirect</strong> API is called).</td>
</tr>
</tbody>
</table>

**RecordStatus** Property Constants (All Are [ ])

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The record has been deleted locally and</td>
<td></td>
</tr>
</tbody>
</table>
dbDBDeleted

in the database.

dbDeleted

The record has been deleted, but not yet deleted in the database.

dbRecordModified

The record has been modified and not updated in the database.

The record has been inserted with the AddNew method, but not yet inserted into the database.

(dbRecordNew)

(Defalut) The record has not been modified or has been updated successfully.

**Type Property Constants**

For any Field, Parameter, or Property object, the Type property may include any of the following (All are or):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbBigInt</td>
<td>Big Integer data (ODBCDirect only)</td>
</tr>
<tr>
<td>dbBinary</td>
<td>Binary data</td>
</tr>
<tr>
<td>dbBoolean</td>
<td>Boolean (True/False) data</td>
</tr>
<tr>
<td>dbByte</td>
<td>Byte (8-bit) data</td>
</tr>
<tr>
<td>dbChar</td>
<td>Character data (ODBCDirect only)</td>
</tr>
<tr>
<td>dbCurrency</td>
<td>Currency data</td>
</tr>
<tr>
<td>dbDate</td>
<td>Date value data</td>
</tr>
<tr>
<td>dbDecimal</td>
<td>Decimal data (ODBCDirect only)</td>
</tr>
<tr>
<td>dbDouble</td>
<td>Double-precision floating-point data</td>
</tr>
<tr>
<td>dbFloat</td>
<td>Floating-point data (ODBCDirect only)</td>
</tr>
<tr>
<td>dbGUID</td>
<td>GUID data</td>
</tr>
<tr>
<td>dbInteger</td>
<td>Integer data</td>
</tr>
<tr>
<td>dbLong</td>
<td>Long Integer data</td>
</tr>
<tr>
<td>dbLongBinary</td>
<td>Binary data (bitmap)</td>
</tr>
<tr>
<td>dbMemo</td>
<td>Memo data (extended text)</td>
</tr>
<tr>
<td>dbNumeric</td>
<td>Numeric data (ODBCDirect only)</td>
</tr>
</tbody>
</table>
**dbSingle**  
Single-precision floating-point data

**dbText**  
Text data (variable width)

**dbTime**  
Data in time format (ODBCDirect only)

**dbTimeStamp**  
Data in time and date format (ODBCDirect only)

**dbVarBinary**  
Variable Binary data (ODBCDirect only)

For any **QueryDef** object, the **Type** property may include any of the following (All are 📜):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbQAction</td>
<td>Action query</td>
</tr>
<tr>
<td>dbQAppend</td>
<td>Append query</td>
</tr>
<tr>
<td>dbQCompound</td>
<td>Compound query (ODBCDirect workspaces only)</td>
</tr>
<tr>
<td>dbQCrosstab</td>
<td>Crosstab query</td>
</tr>
<tr>
<td>dbQDDL</td>
<td>Data-definition language (DDL) query</td>
</tr>
<tr>
<td>dbQDelete</td>
<td>Delete query</td>
</tr>
<tr>
<td>dbQMakeTable</td>
<td>Make-table query</td>
</tr>
<tr>
<td>dbQProcedure</td>
<td>SQL procedure that executes a stored procedure</td>
</tr>
<tr>
<td></td>
<td>(ODBCDirect workspaces only)</td>
</tr>
<tr>
<td>dbQSelect</td>
<td>Select query</td>
</tr>
<tr>
<td>dbQSetOperation</td>
<td>Set operation query</td>
</tr>
<tr>
<td>dbQSPTBulk</td>
<td>Bulk operation query</td>
</tr>
<tr>
<td>dbQSQLPassThrough</td>
<td>SQL pass-through query</td>
</tr>
<tr>
<td>dbQUpdate</td>
<td>Update query</td>
</tr>
</tbody>
</table>

For any **Recordset** object, the **Type** property may include any of the following (All are 📜):

<table>
<thead>
<tr>
<th>Constants</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**UpdateOptions** Property Constants (All Are 🌐)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbCriteriaKey</td>
<td>(Default) Uses just the key column(s) in the where clause.</td>
</tr>
<tr>
<td>dbCriteriaModValues</td>
<td>Uses the key column(s) and all updated columns in the where clause.</td>
</tr>
<tr>
<td>dbCriteriaAllCols</td>
<td>Uses the key column(s) and all the columns in the where clause.</td>
</tr>
<tr>
<td>dbCriteriaTimeStamp</td>
<td>Uses just the timestamp column if available (will generate a run-time error if no timestamp column is in the result set).</td>
</tr>
<tr>
<td>dbCriteriaDeleteInsert</td>
<td>Uses a pair of DELETE and INSERT statements for each modified row.</td>
</tr>
<tr>
<td>dbCriteriaUpdate</td>
<td>(Default) Uses an UPDATE statement for each modified row.</td>
</tr>
</tbody>
</table>

**CompactDatabase, CreateDatabase** Methods Locale Argument Constants (All Are 🌐)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbLangGeneral</td>
<td>English, German, French, Portuguese, Italian, and Modern Spanish</td>
</tr>
<tr>
<td>dbLangArabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>dbLangChineseSimplified</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>Constant</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>dbDecrypt</td>
<td>Decrypts database while compacting</td>
</tr>
<tr>
<td>dbEncrypt</td>
<td>Encrypts database</td>
</tr>
<tr>
<td>dbVersion10</td>
<td>Microsoft Jet database engine version 1.0</td>
</tr>
<tr>
<td>dbVersion11</td>
<td>Microsoft Jet database engine version 1.1</td>
</tr>
<tr>
<td>dbVersion20</td>
<td>Microsoft Jet database engine version 2.0</td>
</tr>
<tr>
<td>dbVersion30</td>
<td>Microsoft Jet database engine version 3.0</td>
</tr>
</tbody>
</table>

**CompactDatabase** Method Options Argument Constants (All Are ™)
### CreateDatabase Method Options Argument Constants (All Are Constants)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbEncrypt</td>
<td>Encrypts database</td>
</tr>
<tr>
<td>dbVersion10</td>
<td>Microsoft Jet database engine version 1.0</td>
</tr>
<tr>
<td>dbVersion11</td>
<td>Microsoft Jet database engine version 1.1</td>
</tr>
<tr>
<td>dbVersion20</td>
<td>Microsoft Jet database engine version 2.0</td>
</tr>
<tr>
<td>dbVersion30</td>
<td>Microsoft Jet database engine version 3.0</td>
</tr>
</tbody>
</table>

### CreateWorkspace Method Type Argument Constants

For any **Workspace** object **Type** property and **DBEngine** object **DefaultType** property, use any of the following: (All Are Constants)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbUseODBC</td>
<td>The next workspace created will use ODBCDirect.</td>
</tr>
<tr>
<td>dbUseJet</td>
<td>The next workspace created will use the Microsoft Jet database engine.</td>
</tr>
</tbody>
</table>

### Execute Method Options Argument Constants (All Are Constants)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbDenyWrite</td>
<td>Denies write permission to other users (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbInconsistent</td>
<td>Allows inconsistent updates (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbConsistent</td>
<td>Allows consistent updates (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbSQLPassThrough</td>
<td>An SQL pass-through. Causes the SQL statement to be passed to an ODBC.</td>
</tr>
</tbody>
</table>
**dbFailOnError**
Rolls back updates if an error occurs (Microsoft Jet workspaces only).

**dbSeeChanges**
Generates a run-time error if another user is changing data you are editing (Microsoft Jet workspaces only).

**dbRunAsync**
Executes the query asynchronously (ODBCDirect workspaces only).

**dbExecDirect**
Executes the query without first calling the SQLPrepare ODBC function (ODBCDirect workspaces only).

---

**Idle Method Optional Argument Constant (This Is ???)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbRefreshCache</td>
<td>Forces any pending writes to disk, and refreshes memory from current disk files.</td>
</tr>
</tbody>
</table>

---

**MakeReplica Method Optional Argument Constants (All are ???)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbRepMakePartial</td>
<td>Creates a partial replica.</td>
</tr>
<tr>
<td>dbRepMakeReadOnly</td>
<td>Makes replicable elements of new database read-only.</td>
</tr>
</tbody>
</table>

---

**OpenConnection and OpenDatabase Methods Option Argument Constants (All Are ???)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbDriverNoPrompt</td>
<td>The driver manager uses the connection string provided in connect. If sufficient information is not provided, a trappable</td>
</tr>
</tbody>
</table>
error is returned.
The driver manager displays the ODBC Data Sources dialog box. The connection string used to establish the connection is constructed from the data source name (DSN) selected and completed by the user via the dialog boxes.

If the connection string provided includes the DSN keyword, the driver manager uses the string as provided in connect, otherwise it behaves as it does when dbDriverPrompt is specified.

(Default) Behaves like dbDriverComplete except the driver disables the controls for any information not required to complete the connection.

OpenRecordset Method Type Argument Constants (All Are 🔄)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbOpenDynamic</td>
<td>Opens a dynamic-type Recordset (ODBCDirect workspaces only)</td>
</tr>
<tr>
<td>dbOpenDynaset</td>
<td>Opens a dynaset-type Recordset</td>
</tr>
<tr>
<td>dbOpenForwardOnly</td>
<td>Opens a forward-only type Recordset</td>
</tr>
<tr>
<td>dbOpenSnapshot</td>
<td>Opens a snapshot-type Recordset</td>
</tr>
<tr>
<td>dbOpenTable</td>
<td>Opens a table-type Recordset (Microsoft Jet workspaces only)</td>
</tr>
</tbody>
</table>

OpenRecordset Method LockEdits Argument Constants (All Are 🔄)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbPessimistic</td>
<td>Pessimistic concurrency. Cursor uses the lowest level of locking sufficient to</td>
</tr>
</tbody>
</table>
ensure the record can be updated.

**dbReadOnly**

Cursor is read-only. No updates are allowed.

**dbOptimistic**

Optimistic concurrency based on record ID. Cursor compares record ID in old and new records to determine if changes have been made since the record was last accessed.

**dbOptimisticValue**

Optimistic concurrency based on record values. Cursor compares data values in old and new records to determine if changes have been made since the record was last accessed ([ODBCDirect workspaces](#) only).

**dbOptimisticBatch**

Enables batch optimistic updates (ODBCDirect workspaces only).

---

**OpenRecordset Method Options Argument Constants (All Are `int`)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbDenyWrite</td>
<td>Prevents other users from changing Recordset records (<a href="#">Microsoft Jet workspaces</a> only).</td>
</tr>
<tr>
<td>dbDenyRead</td>
<td>Prevents other users from reading Recordset records (table-type in Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbReadOnly</td>
<td>Opens the Recordset as read-only (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbAppendOnly</td>
<td>Allows user to add new records to the dynaset, but prevents user from reading existing records (dynaset-type in Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbInconsistent</td>
<td>Applies updates to all dynaset fields, even if other records are affected (dynaset- and snapshot-type in Microsoft Jet workspaces only).</td>
</tr>
</tbody>
</table>
**dbConsistent**

Applies updates only to those fields that will not affect other records in the dynaset (dynaset- and snapshot-type in Microsoft Jet workspaces only).

**dbSQLPassThrough**

Sends an SQL statement to an ODBC database (snapshot-type in Microsoft Jet workspaces only).

**dbForwardOnly**

Creates a forward-only scrolling snapshot-type Recordset (snapshot-type in Microsoft Jet workspaces only).

**dbSeeChanges**

Generates a run-time error if another user is changing data you are editing (dynaset-type in Microsoft Jet workspaces only).

**dbRunAsync**

Executes the query asynchronously (ODBCDirect workspaces only).

**dbExecDirect**

Executes the query without first calling the SQLPrepare ODBC function (ODBCDirect workspaces only).

---

**SetOption** Method Parameter Constants (All Are **qi**)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbPageTimeout</td>
<td>The PageTimeout key</td>
</tr>
<tr>
<td>dbSharedAsyncDelay</td>
<td>The SharedAsyncDelay key</td>
</tr>
<tr>
<td>dbExclusiveAsyncDelay</td>
<td>The ExclusiveAsyncDelay key</td>
</tr>
<tr>
<td>dbLockRetry</td>
<td>The LockRetry key</td>
</tr>
<tr>
<td>dbUserCommitSync</td>
<td>The UserCommitSync key</td>
</tr>
<tr>
<td>dbImplicitCommitSync</td>
<td>The ImplicitCommitSync key</td>
</tr>
<tr>
<td>dbMaxBufferSize</td>
<td>The MaxBufferSize key</td>
</tr>
<tr>
<td>dbMaxLocksPerFile</td>
<td>The MaxLocksPerFile key</td>
</tr>
<tr>
<td>dbLockDelay</td>
<td>The LockDelay key</td>
</tr>
<tr>
<td>dbRecycleLVs</td>
<td>The RecycleLVs key</td>
</tr>
<tr>
<td>dbFlushTransactionTimeout</td>
<td>The FlushTransactionTimeout key</td>
</tr>
</tbody>
</table>
### Synchronize Method Exchange Argument Constants (All Are `const`)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbRepExportChanges</td>
<td>Sends changes from current database to target database.</td>
</tr>
<tr>
<td>dbRepImportChanges</td>
<td>Receives changes from target database.</td>
</tr>
<tr>
<td>dbRepImpExpChanges</td>
<td>Sends and receives data in a bidirectional exchange.</td>
</tr>
<tr>
<td>dbRepSyncInternet</td>
<td>Exchanges data between files connected via an Internet pathway.</td>
</tr>
</tbody>
</table>

### Update Method Type Argument Constants (All Are `const`)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbUpdateRegular</td>
<td>(Default) Pending changes aren't cached and are written to disk immediately.</td>
</tr>
<tr>
<td>dbUpdateBatch</td>
<td>All pending changes in the update cache are written to disk.</td>
</tr>
<tr>
<td>dbUpdateCurrentRecord</td>
<td>Only the current record's pending changes are written to disk.</td>
</tr>
</tbody>
</table>

### CancelUpdate Method Type Argument Constants (All Are `const`)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbUpdateRegular</td>
<td>(Default) Pending changes aren't cached and are written to disk immediately.</td>
</tr>
<tr>
<td>dbUpdateBatch</td>
<td>All pending changes in the update cache are written to disk.</td>
</tr>
</tbody>
</table>
DAO objects and collections provide a framework for using code to create and manipulate components of your database system. Objects and collections have properties that describe the characteristics of database components and methods that you use to manipulate them. Together these objects and collections form a hierarchical model of your database structure, which you can control programmatically.

Objects and collections provide different types of containment relations: Objects contain zero or more collections, all of different types; and collections contain zero or more objects, all of the same type. Although objects and collections are similar entities, the distinction differentiates the two types of relations.

In the following table, the type of collection in the first column contains the type of object in the second column. The third column describes what each type of object represents.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connections</td>
<td>Connection</td>
<td>Information about a connection to an ODBC data source ([ODBCDirect workspaces](ODBCDirect workspaces) only)</td>
</tr>
<tr>
<td>Containers</td>
<td>Container</td>
<td>Storage for information about a predefined object type ([Microsoft Jet](Microsoft Jet))</td>
</tr>
<tr>
<td><strong>Databases</strong></td>
<td><strong>Database</strong></td>
<td>workspaces only</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>None</td>
<td>DBEngine</td>
<td>Microsoft Jet database engine</td>
</tr>
</tbody>
</table>

| **Documents** | **Document** |  |
|---------------|--------------|  |

<table>
<thead>
<tr>
<th><strong>Errors</strong></th>
<th><strong>Error</strong></th>
<th>Information about any errors associated with this object</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Fields</strong></th>
<th><strong>Field</strong></th>
<th>A column that is part of a table, query, index, relation, or recordset</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Groups</strong></th>
<th><strong>Group</strong></th>
<th>A group of user accounts (Microsoft Jet workspaces only)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Indexes</strong></th>
<th><strong>Index</strong></th>
<th>Predefined ordering and uniqueness of values in a table (Microsoft Jet workspaces only)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Parameters</strong></th>
<th><strong>Parameter</strong></th>
<th>A parameter for a parameter query</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Properties</strong></th>
<th><strong>Property</strong></th>
<th>A built-in or user-defined property</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>QueryDefs</strong></th>
<th><strong>QueryDef</strong></th>
<th>A saved query definition</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Recordsets</strong></th>
<th><strong>Recordset</strong></th>
<th>The records in a base table or query</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Relations</strong></th>
<th><strong>Relation</strong></th>
<th>A relationship between fields in tables and queries (Microsoft Jet workspaces only)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>TableDefs</strong></th>
<th><strong>TableDef</strong></th>
<th>A saved table definition (Microsoft Jet workspaces only)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Users</strong></th>
<th><strong>User</strong></th>
<th>A user account (Microsoft Jet workspaces only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workspaces</td>
<td>Workspace</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td></td>
</tr>
</tbody>
</table>

Jet workspaces only)

A **session** of the Microsoft Jet database engine
DAO Object Model for Microsoft Jet Workspaces
DAO Object Model for ODBC Direct Workspaces
A **Connection** object represents a connection to an **ODBC** database (**ODBCDirect workspaces** only).

**Remarks**

A **Connection** is a non-persistent object that represents a connection to a remote database. The **Connection** object is only available in ODBCDirect workspaces (that is, a **Workspace** object created with the type option set to **dbUseODBC**).
Note  Code written for earlier versions of DAO can continue to use the Database object for backward compatibility, but if the new features of a Connection are desired, you should revise code to use the Connection object. To help with code conversion, you can obtain a Connection object reference from a Database by reading the Connection property of the Database object. Conversely, you can obtain a Database object reference from the Connection object’s Database property.
A **Connections** collection contains the current **Connection** objects of a **Workspace** object. ([ODBCDirect workspaces](#) only).

**Remarks**

When you open a **Connection** object, it is automatically appended to the **Connections** collection of the **Workspace**. When you close a **Connection** object with the **Close** method, it is removed from the **Connections** collection.
You should close all open Recordset objects within the Connection before closing it.

At the same time you open a Connection object, a corresponding Database object is created and appended to the Databases collection in the same Workspace, and vice versa. Similarly, when you close the Connection, the corresponding Database is deleted from the Databases collection, and so on.

The Name property setting of a Connection is a string that specifies the path of the database file. To refer to a Connection object in a collection by its ordinal number or by its Name property setting, use any of the following syntax forms:

`Connections(0)`

`Connections("name")`

`Connections![name]`

Note You can open the same data source more than once, creating duplicate names in the Connections collection. You should assign Connection objects to object variables and refer to them by variable name.
Connection Object, Connections
Collection Summary

Connection Object

The Connection object contains these collections, methods, and properties.

Collections

QueryDefs (default)
Recordsets

Methods

Cancel
Close
CreateQueryDef
Execute
OpenRecordset

Properties

Connect
Database
Name
Connections Collection

A Connections collection is contained in each ODBCDirect Workspace object, and contains this method and this property:

Method

Refresh

Property

Count
Containers Collection

A Containers collection contains all of the Container objects that are defined in a database (Microsoft Jet databases only).

Remarks

Each Database object has a Containers collection consisting of built-in Container objects. Some of these Container objects are defined by the Microsoft Jet database engine while others may be defined by other applications.
Container Object

A Container object groups similar types of Document objects together.

Remarks

Each Database object has a Containers collection consisting of built-in Container objects. Applications can define their own document types and corresponding containers (Microsoft Jet databases only); however, these objects may not always be supported through DAO.
Some of these **Container** objects are defined by the [Microsoft Jet database engine](https://docs.microsoft.com/en-us/sql/jet/jet-engine) while others may be defined by other applications. The following table lists the name of each **Container** object defined by the Microsoft Jet database engine and what type of information it contains.

<table>
<thead>
<tr>
<th>Container name</th>
<th>Contains information about</th>
</tr>
</thead>
<tbody>
<tr>
<td>Databases</td>
<td>Saved databases</td>
</tr>
<tr>
<td>Tables</td>
<td>Saved tables and queries</td>
</tr>
<tr>
<td>Relations</td>
<td>Saved relationships</td>
</tr>
</tbody>
</table>

**Note** Don't confuse the **Container** objects listed in the preceding table with the collections of the same name. The Databases **Container** object refers to all saved database objects, but the Databases collection refers only to database objects that are open in a particular workspace.

Each **Container** object has a **Documents** collection containing **Document** objects that describe instances of built-in objects of the type specified by the **Container**. You typically use a **Container** object as an intermediate link to the information in the **Document** object. You can also use the **Containers** collection to set security for all **Document** objects of a given type.

With an existing **Container** object, you can:

- Use the **Name** property to return the predefined name of the **Container** object.

- Use the **Owner** property to set or return the owner of the **Container** object. To set the **Owner** property, you must have write permission for the **Container** object, and you must set the property to the name of an existing **User** or **Group** object.

- Use the **Permissions** and **UserName** properties to set access permissions for the **Container** object; any **Document** object created in the **Documents** collection of a **Container** object inherits these access permission settings.

Because **Container** objects are built-in, you can't create new **Container** objects or delete existing ones.
To refer to a **Container** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

- `Containers(0)`
- `Containers("name")`
- `Containers![name]`
Container Object, Containers Collection Summary

Container Object

A Container object contains no methods; it contains these collections and properties.

Collections

Documents (Default)
Properties

Properties

AllPermissions
Inherit
Name
Owner
Permissions
UserName

Containers Collection

A Containers collection appears in each Database object of a Microsoft Jet database, and contains this method and this property.
A **Databases** collection contains all open **Database** objects opened or created in a **Workspace** object.

Remarks
When you open an existing **Database** object or create a new one from a **Workspace**, it is automatically appended to the **Databases** collection. When you close a **Database** object with the **Close** method, it is removed from the **Databases** collection but not deleted from disk. You should close all open **Recordset** objects before closing a **Database** object.

In a Microsoft Jet workspace, the **Name** property setting of a database is a string that specifies the path of the database file. In an ODBCDirect workspace, the **Name** property is the name of the corresponding **Connection** object.

To refer to a **Database** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

**Databases(0)**

**Databases("name")**

**Databases![name]**

**Note**  You can open the same data source or database more than once, creating duplicate names in the **Databases** collection. You should assign **Database** objects to **object variables** and refer to them by variable name.
A **Database** object represents an open database.

**Remarks**
You use the **Database** object and its methods and properties to manipulate an open database. In any type of database, you can:

- Use the **Execute** method to run an *action query*.
- Set the **Connect** property to establish a connection to an ODBC data source.
- Set the **QueryTimeout** property to limit the length of time to wait for a query to execute against an ODBC data source.
- Use the **RecordsAffected** property to determine how many records were changed by an action query.
- Use the **OpenRecordset** method to execute a *select query* and create a **Recordset** object.
- Use the **Version** property to determine which version of a database engine created the database.

With a Microsoft Jet database (.mdb file), you can also use other methods, properties, and collections to manipulate a **Database** object, as well as create, modify, or get information about its tables, queries, and relationships. For example, you can:

- Use the **CreateTableDef** and **CreateRelation** methods to create tables and relations, respectively.
- Use the **CreateProperty** method to define new **Database** properties.
- Use the **CreateQueryDef** method to create a *persistent* or temporary query definition.
- Use **MakeReplica, Synchronize, and PopulatePartial** methods to create and synchronize full or partial **replicas** of your database.
- Set the **CollatingOrder** property to establish the alphabetic sorting order for character-based fields in different languages.
In an ODBCDirect workspace, you can:

- Use the **Connection** property to obtain a reference to the **Connection** object that corresponds to the **Database** object.

  **Note** For a complete list of all methods, properties, and collections available on a **Database** object in either a Microsoft Jet workspace or ODBCDirect workspace, see the [Summary](#) topic.

You use the **CreateDatabase** method to create a persistent **Database** object that is automatically appended to the **Databases** collection, thereby saving it to disk.

You don't need to specify the **DBEngine** object when you use the **OpenDatabase** method.

Opening a database with linked tables doesn't automatically establish links to the specified external files or Microsoft Jet-connected ODBC data sources. You must either reference the table's **TableDef** or **Field** objects or open a **Recordset** object. If you can't establish links to these tables, a trappable error occurs. You may also need permission to access the database, or another user might have the database opened exclusively. In these cases, trappable errors occur.

You can also use the **OpenDatabase** method to open an external database (such as Excel, dBASE, and Paradox) directly instead of opening a Microsoft Jet database that has links to its tables.

**Note** Opening a **Database** object directly on a Microsoft Jet-connected ODBC data source, such as Microsoft SQL Server, is not recommended because query performance is much slower than when using linked tables. However, performance is not a problem with opening a **Database** object directly on an external ISAM database file, such as Excel, Paradox, and so forth.

When a procedure that declares a **Database** object has executed, local **Database** objects are closed along with any open **Recordset** objects. Any pending updates are lost and any pending transactions are rolled back, but no trappable error occurs. You should explicitly complete any pending transactions or edits and close **Recordset** objects and **Database** objects before exiting procedures that declare these object variables locally.

When you use one of the transaction methods (**BeginTrans**, **CommitTrans**, or
**Rollback** on the *Workspace* object, these *transactions* apply to all databases opened on the *Workspace* from which the *Database* object was opened. If you want to use independent transactions, you must first open an additional *Workspace* object, and then open another *Database* object in that *Workspace* object.

**Note**  You can open the same data source or database more than once, creating duplicate names in the *Databases* collection. You should assign *Database* objects to *object variables* and refer to them by variable name.
Database Object, Databases
Collection Summary

Database Object

A Database object contains these collections, methods, and properties.

Legend:

Feature available in Microsoft Jet workspaces only.

Feature available in ODBCDirect workspaces only.

Collections

Containers  Properties
QueryDefs  Recordsets (Default for DAD ODBC)
Relations  TableDefs (Default for)

Methods

Close
CreateProperty  CreateQueryDef
Properties

CollatingOrder  
Connect  
Connection  DAD  ODBC  
DesignMasterID  
Name  
QueryTimeout  
RecordsAffected  
Replicable  (user-defined)  
ReplicaID  
Updatable  
V1xNullBehavior  
Version

Databases Collection

A Databases collection appears in each Workspace object, and contains this method and this property.

Method

Refresh

Property
Count
DBEngine Object

The **DBEngine** object is the top level object in the **DAO** object model.

**Remarks**

The **DBEngine** object contains and controls all other objects in the hierarchy of DAO objects. You can't create additional **DBEngine** objects, and the **DBEngine** object isn't an element of any collection.

**Note**  When you reference an **ODBC data source** directly through DAO, it is called an "**ODBCDirect workspace**." This is to distinguish it from an ODBC data...
source that you reference indirectly through the Microsoft Jet database engine, using a "Microsoft Jet workspace." Each method of accessing ODBC data requires one of two types of Workspace object; you can set the DefaultType property to choose the default type of Workspace object that you will create from the DBEngine object. The Workspace type and associated data source determines which DAO objects, methods, and properties you can use.

With any type of database or connection, you can:

- Use the Version property to obtain the DAO version number.

- Use the LoginTimeout property to obtain or set the ODBC login timeout, and the RegisterDatabase method to provide ODBC information to the Microsoft Jet database engine. You can use these features the same way, regardless of whether you connect to the ODBC data source through Microsoft Jet or through an ODBCDirect workspace.

- Use the DefaultType property to set the default type of database connection that subsequently created Workspace objects will use — either Microsoft Jet or ODBCDirect.

- Use the DefaultPassword and DefaultUser properties to set the user identification and password for the default Workspace object.

- Use the CreateWorkspace method to create a new Workspace object. You can use optional arguments to override the settings of the DefaultType, DefaultPassword, and DefaultUser properties.

- Use the OpenDatabase method to open a database in the default Workspace, and use the BeginTrans, Commit, and Rollback methods to control transactions on the default Workspace.

- Use the Workspaces collection to reference specific Workspace objects.

- Use the Errors collection to examine data access error details.

Other properties and methods are only available when you use DAO with the Microsoft Jet database engine. You can use them to control the Microsoft Jet database engine, manipulate its properties, and perform tasks on temporary
objects that aren't elements of collections. For example, you can:

- Use the **CreateDatabase** method to create a new Microsoft Jet **Database** object.

- Use the **Idle** method to enable the Microsoft Jet database engine to complete any pending tasks.

- Use the **CompactDatabase** and **RepairDatabase** methods to maintain database files.

- Use the **IniPath** and **SystemDB** properties to specify the location of Microsoft Jet Windows Registry information and the Microsoft Jet workgroup information file, respectively. The **SetOption** method allows you override windows registry settings for the Microsoft Jet database engine.

After you change the **DefaultType** and **IniPath** property settings, only subsequent **Workspace** objects will reflect these changes.

**Note**  For a complete list of all methods, properties, and collections available on the **DBEngine** object, see the **Summary** topic.

To refer to a collection that belongs to the **DBEngine** object, or to refer to a method or property that applies to this object, use this syntax:

```
[DBEngine.][collection | method | property]
```
DBEngine Object Summary

The DBEngine object contains these collections, methods, and properties.

Legend:

- Feature available in Microsoft Jet workspaces only.

- Feature available in ODBC Direct workspaces only.

Collections

- Errors
- Properties
- Workspaces (Default)

Methods

- BeginTrans
- CommitTrans
- CompactDatabase
- CreateDatabase
- CreateWorkspace
- Idle
- OpenConnection
- OpenDatabase
- RegisterDatabase
RepairDatabase
Rollback
SetOption

Properties

DefaultPassword
DefaultType
DefaultUser
IniPath
LoginTimeout
SystemDB
Version
Documents Collection

A Documents collection contains all of the Document objects for a specific type of object (Microsoft Jet databases only).

Remarks

Each Container object has a Documents collection containing Document objects that describe instances of built-in objects of the type specified by the Container.

To refer to a Document object in a collection by its ordinal number or by its
Name property setting, use any of the following syntax forms:

Documents(0)

Documents("name")

Documents![name]
A **Document** object includes information about one instance of an object. The object can be a database, saved table, query, or relationship (Microsoft Jet databases only).

**Remarks**

Each **Container** object has a **Documents** collection containing **Document** objects that describe instances of built-in objects of the type specified by the **Container**. The following table lists the type of object each **Document** describes, the name of its **Container** object, and what type of information
**Document** contains.

<table>
<thead>
<tr>
<th>Document</th>
<th>Container</th>
<th>Contains information about</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Databases</td>
<td>Saved database</td>
</tr>
<tr>
<td>Table or query</td>
<td>Tables</td>
<td>Saved table or query</td>
</tr>
<tr>
<td>Relationship</td>
<td>Relations</td>
<td>Saved relationship</td>
</tr>
</tbody>
</table>

**Note**  Don't confuse the **Container** objects listed in the preceding table with the collections of the same name. The Databases **Container** object refers to all saved database objects, but the **Databases** collection refers only to database objects that are open in a particular workspace.

With a **Document** object, you can:

- Use the **Name** property to return the name that a user or the **Microsoft Jet database engine** gave to the object when it was created.

- Use the **Container** property to return the name of the **Container** object that contains the **Document** object.

- Use the **Owner** property to set or return the owner of the object. To set the **Owner** property, you must have write permission for the **Document** object, and you must set the property to the name of an existing **User** or **Group** object.

- Use the **UserName** or **Permissions** properties to set or return the access permissions of a user or group for the object. To set these properties, you must have write permission for the **Document** object, and you must set the **UserName** property to the name of an existing **User** or **Group** object.

- Use the **DateCreated** and **LastUpdated** properties to return the date and time when the **Document** object was created and last modified.

Because a **Document** object corresponds to an existing object, you can't create new **Document** objects or delete existing ones. To refer to a **Document** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:
Documents(0)

Documents("name")

Documents![name]
Document Object, Documents Collection Summary

Document Object

A Document object contains this collection, this method, and these properties.

Collection

Properties

Method

CreateProperty

Properties

AllPermissions
Container
DateCreated
KeepLocal (user-defined)
LastUpdated
Name
Owner
Permissions
Replicable (user-defined)
UserName
**Documents Collection**

A [Documents] collection appears in each [Container] object, and contains this method and this property.

**Method**

**Refresh**

**Property**

**Count**
This **Recordset** type represents a query result set from one or more base tables in which you can add, change, or delete records from a row-returning query. Further, records that other users add, delete, or edit in the base tables also appear in your **Recordset**.

This type is only available in [ODBCDirect workspaces](http://example.com), and corresponds to an [ODBC dynamic cursor](http://example.com).
Dynaset-Type Recordset Object Summary

The `dynaset-type Recordset` object contains these collections, methods, and properties.

**Legend:**

- **Feature available in Microsoft Jet workspaces only.**
- **Feature available in ODBC Direct workspaces only.**

**Collections**

- **Fields** (default)
- **Properties**

**Methods**

- `AddNew`
- `Cancel`
- `CancelUpdate`
- `Clone`
- `Close`
- `CopyQueryDef`
- `Delete`
- `Edit`
Properties

The following table indicates whether the property setting is read/write, read-only, or only available in either Microsoft Jet or ODBCDirect workspaces.

- Read-only
- Read/write

<table>
<thead>
<tr>
<th>Properties</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsolutePosition</td>
<td></td>
</tr>
<tr>
<td>BatchCollisionCount</td>
<td>DAO, ODBC</td>
</tr>
<tr>
<td>BatchCollisions</td>
<td>DAO, ODBC</td>
</tr>
<tr>
<td>BatchSize</td>
<td>DAO, ODBC</td>
</tr>
<tr>
<td>BOF</td>
<td></td>
</tr>
<tr>
<td>Bookmark</td>
<td></td>
</tr>
<tr>
<td>Bookmarkable</td>
<td></td>
</tr>
</tbody>
</table>

Microsoft Jet
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheSize</td>
<td></td>
</tr>
<tr>
<td>CacheStart</td>
<td></td>
</tr>
<tr>
<td>Connection</td>
<td></td>
</tr>
<tr>
<td>EditMode</td>
<td></td>
</tr>
<tr>
<td>EOF</td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td></td>
</tr>
<tr>
<td>LastModified</td>
<td></td>
</tr>
<tr>
<td>LockEdits</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>NoMatch</td>
<td></td>
</tr>
<tr>
<td>PercentPosition</td>
<td></td>
</tr>
<tr>
<td>RecordCount</td>
<td></td>
</tr>
<tr>
<td>RecordStatus</td>
<td></td>
</tr>
<tr>
<td>Restartable</td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td></td>
</tr>
<tr>
<td>StillExecuting</td>
<td></td>
</tr>
<tr>
<td>Transactions</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>Updatable</td>
<td></td>
</tr>
<tr>
<td>UpdateOptions</td>
<td></td>
</tr>
<tr>
<td>ValidationRule</td>
<td></td>
</tr>
<tr>
<td>ValidationText</td>
<td></td>
</tr>
</tbody>
</table>
Errors Collection

An **Errors** collection contains all stored **Error** objects, each of which pertains to a single operation involving **DAO**.

![Diagram](image)

**Remarks**

Any operation involving DAO objects can generate one or more **Error** objects. As each error occurs, one or more **Error** objects are placed in the **Errors** collection of the **DBEngine** object. When another DAO operation generates an error, the **Errors** collection is cleared, and the new set of **Error** objects is placed in the **Errors** collection. The highest-numbered object in the **Errors** collection
(DBEngine.Errors.Count - 1) corresponds to the error reported by the Microsoft Visual Basic for Applications (VBA) `Err` object.

DAO operations that don't generate an error have no effect on the `Errors` collection.

Elements of the `Errors` collection aren't appended as they typically are with other collections, so the `Errors` collection doesn't support the `Append` and `Delete` methods.

The set of `Error` objects in the `Errors` collection describes one error. The first `Error` object is the lowest level error, the second the next higher level, and so forth. For example, if an ODBC error occurs while trying to open a Recordset object, the first error object contains the lowest level ODBC error; subsequent errors contain the ODBC errors returned by the various layers of ODBC. In this case, the ODBC driver manager, and possibly the driver itself, return separate `Error` objects. The last `Error` object contains the DAO error indicating that the object couldn't be opened.

Enumerating the specific errors in the `Errors` collection enables your error-handling routines to more precisely determine the cause and origin of an error, and take appropriate steps to recover.

**Note** If you use the `New` keyword to create an object that causes an error either before or while being placed into the `Errors` collection, the collection doesn't contain error information about that object, because the new object is not associated with the `DBEngine` object. However, the error information is available in the VBA `Err` object.
An **Error** object contains details about data access errors, each of which pertains to a single operation involving **DAO**.

![Diagram showing the relationship between DBEngine, Errors, and Error objects]

**Remarks**

Any operation involving DAO can generate one or more errors. For example, a call to an **ODBC** server might result in an error from the database server, an error from ODBC, and a DAO error. As each such error occurs, an **Error** object is placed in the **Errors** collection of the **DBEngine** object. A single event can therefore result in several **Error** objects appearing in the **Errors** collection.
When a subsequent DAO operation generates an error, the **Errors** collection is cleared, and one or more new **Error** objects are placed in the **Errors** collection. DAO operations that don't generate an error have no effect on the **Errors** collection.

The set of **Error** objects in the **Errors** collection describes one error. The first **Error** object is the lowest level error (the originating error), the second the next higher level error, and so forth. For example, if an ODBC error occurs while trying to open a **Recordset** object, the first **Error** object — **Errors**(0) — contains the lowest level ODBC error; subsequent errors contain the ODBC errors returned by the various layers of ODBC. In this case, the ODBC driver manager, and possibly the driver itself, return separate **Error** objects. The last **Error** object — **Errors.Count**-1 — contains the DAO error indicating that the object couldn't be opened.

Enumerating the specific errors in the **Errors** collection enables your error-handling routines to more precisely determine the cause and origin of an error, and take appropriate steps to recover. On both Microsoft Jet and ODBCDirect workspaces, you can read the **Error** object’s properties to obtain specific details about each error, including:

- The **Description** property, which contains the text of the error alert that will be displayed on the screen if the error is not trapped.

- The **Number** property, which contains the **Long** integer value of the error constant.

- The **Source** property, which identifies the object that raised the error. This is particularly useful when you have several **Error** objects in the **Errors** collection following a request to an ODBC data source.

- The **HelpFile** and **HelpContext** properties, which indicate the appropriate Microsoft Windows Help file and Help topic, respectively, (if any exist) for the error.

**Note** When programming in Microsoft Visual Basic for Applications (VBA), if you use the **New** keyword to create an object that subsequently causes an error before that object has been appended to a collection, the **DBEngine** object's **Errors** collection won't contain an entry for that
object's error, because the new object is not associated with the DBEngine object. However, the error information is available in the VBA Err object.

Your VBA error-handling code should examine the Errors collection whenever you anticipate a data access error. If you are writing a centralized error handler, test the VBA Err object to determine if the error information in the Errors collection is valid. If the Number property of the last element of the Errors collection (DBEngine.Errors.Count - 1) and the value of the Err object match, you can then use a series of Select Case statements to identify the particular DAO error or errors that occurred. If they do not match, use the Refresh method on the Errors collection.
Error Object, Errors Collection

Summary

Error Object

An Error object contains no methods; no collections, and these properties:

Properties

Description
HelpContext
HelpFile
Number
Source

Errors Collection

An Errors collection appears in the DBEngine object, and contains this method and this property:

Method
Refresh

Property
Count
A **Fields** collection contains all stored **Field** objects of an **Index**, **QueryDef** (Microsoft Jet workspaces only), **Recordset**, **Relation**, or **TableDef** object.

**Remarks**

The **Fields** collections of the **Index**, **QueryDef**, **Relation**, and **TableDef** objects
contain the specifications for the fields those objects represent. The **Fields** collection of a **Recordset** object represents the **Field** objects in a row of data, or in a record. You use the **Field** objects in a **Recordset** object to read and to set values for the fields in the **current record** of the **Recordset** object.

To refer to a **Field** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

**Fields**(0)

**Fields**("name")

**Fields**![name]

With the same syntax forms, you can also refer to the **Value** property of a **Field** object that you create and append to a **Fields** collection. The context of the field reference will determine whether you are referring to the **Field** object or the **Value** property of the **Field** object.
Field Object

A Field object represents a column of data with a common data type and a common set of properties.

Remarks

The Fields collections of Index, QueryDef, Relation, and TableDef objects
contain the specifications for the fields those objects represent. The **Fields**
collection of a **Recordset** object represents the **Field** objects in a row of data, or
in a record. You use the **Field** objects in a **Recordset** object to read and set
values for the fields in the **current record** of the **Recordset** object.

In both **Microsoft Jet** and **ODBCDirect** workspaces, you manipulate a field using
a **Field** object and its methods and properties. For example, you can:

- Use the **OrdinalPosition** property to set or return the presentation order of
  the **Field** object in a **Fields** collection. (This property is read-only for
  ODBCDirect databases.)

- Use the **Value** property of a field in a **Recordset** object to set or return
  stored data.

- Use the **AppendChunk** and **GetChunk** methods and the **FieldSize**
  property to get or set a value in an **OLE Object** or **Memo** field of a
  **Recordset** object.

- Use the **Type**, **Size**, and **Attributes** properties to determine the type of data
  that can be stored in the field.

- Use the **SourceField** and **SourceTable** properties to determine the original
  source of the data.

In **Microsoft Jet** workspaces, you can:

- Use the **ForeignName** property to set or return information about a foreign
  field in a **Relation** object.

- Use the **AllowZeroLength**, **DefaultValue**, **Required**, **ValidateOnSet**,
  **ValidationRule**, or **ValidationText** properties to set or return validation
  conditions.

- Use the **DefaultValue** property of a field on a **TableDef** object to set the
  default value for this field when new records are added.

In **ODBCDirect** workspaces, you can:
• Use the Value, VisibleValue, and OriginalValue properties to verify successful completion of a batch update.

**Note** For a complete list of all methods, properties, and collections available on a Field object in any database or connection, see the Summary topic.

To create a new Field object in an Index, TableDef, or Relation object, use the CreateField method.

When you access a Field object as part of a Recordset object, data from the current record is visible in the Field object's Value property. To manipulate data in the Recordset object, you don't usually reference the Fields collection directly; instead, you indirectly reference the Value property of the Field object in the Fields collection of the Recordset object.

To refer to a Field object in a collection by its ordinal number or by its Name property setting, use any of the following syntax forms:

`Fields(0)`

`Fields("name")`

`Fields![name]`

With the same syntax forms, you can also refer to the Value property of a Field object that you create and append to a Fields collection. The context of the field reference will determine whether you are referring to the Field object or the Value property of the Field object.
Field Object, Fields Collection Summary

Field Object

A Field object contains this collection, these methods, and these properties.

Legend:

Feature available in Microsoft Jet workspaces only.

Feature available in ODBCDirect workspaces only.

Collection

Properties

Methods

The following table lists all of the Field object methods. The type of object whose Fields collection contains the Field object determines which methods are available.

<table>
<thead>
<tr>
<th>Method</th>
<th>Index</th>
<th>QueryDef</th>
<th>Recordset</th>
<th>Relation</th>
<th>TableDef</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppendChunk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CreateProperty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetChunk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Properties

The following table lists all of the **Field** object properties. The type of object whose **Fields** collection contains the **Field** object determines which properties are available. All properties are read-only for **Field** objects appended to **Fields** collections of **Index**, **Relation**, and **TableDef** objects.

- **Read-only**
- **Read/write**

<table>
<thead>
<tr>
<th>Property</th>
<th>Index</th>
<th>QueryDef</th>
<th>Recordset</th>
<th>Relation</th>
<th>TableDef</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllowZeroLength</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Attributes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CollatingOrder</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DataUpdatable</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DefaultValue</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FieldSize</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ForeignName</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Name</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OrdinalPosition</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OriginalValue</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Required</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SourceField</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SourceTable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValidateOnSet</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValidationRule</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValidationText</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VisibleValue</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* These properties are only available in an ODBCDirect workspace whose **DefaultCursorDriver** property is set to **dbUseClientBatchCursor**.
## Fields Collection

A **Fields** collection appears in each of the **TableDef**, **QueryDef**, **Recordset**, **Relation**, and **Index** objects, and contains these methods and this property.

<table>
<thead>
<tr>
<th>Method</th>
<th>Index</th>
<th>QueryDef</th>
<th>Recordset</th>
<th>Relation</th>
<th>TableDef</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Delete</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Refresh</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Index</th>
<th>QueryDef</th>
<th>Recordset</th>
<th>Relation</th>
<th>TableDef</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Forward-Only–Type Recordset Object

This Recordset type is identical to a snapshot except that you can only scroll forward through its records. This improves performance in situations where you only need to make a single pass through a result set.

In an ODBCDirect workspace, this type corresponds to an ODBC forward-only cursor.
Forward-Only–Type Recordset Object Summary

The forward-only type Recordset object contains these collections, methods, and properties.

Legend:

- Feature available in Microsoft Jet workspaces only.

- Feature available in ODBCDirect workspaces only.

Collections

Fields (default)
Properties

<table>
<thead>
<tr>
<th>Methods</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddNew</td>
<td>DAD - ODBC</td>
</tr>
<tr>
<td>Cancel</td>
<td>DAD - ODBC</td>
</tr>
<tr>
<td>CancelUpdate</td>
<td>DAD - ODBC</td>
</tr>
<tr>
<td>Close</td>
<td>DAD - ODBC</td>
</tr>
<tr>
<td>CopyQueryDef</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>DAD - ODBC</td>
</tr>
<tr>
<td>Edit</td>
<td></td>
</tr>
<tr>
<td>GetRows</td>
<td>DAD - ODBC</td>
</tr>
</tbody>
</table>
**Move**

Only with forward moves that don't use a bookmark offset.

**MoveNext**

**NextRecordset**

**Requery**

**Update**

**Properties**

The following table indicates whether each property setting is read/write, read-only, or always **False** in either Microsoft Jet or ODBCDirect workspaces.

- **Read-only**
- **Read/write**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatchCollisionCount</td>
<td>DAD, ODBC</td>
</tr>
<tr>
<td>BatchCollisions</td>
<td>DAD, ODBC</td>
</tr>
<tr>
<td>BatchSize</td>
<td>DAD, ODBC</td>
</tr>
<tr>
<td>BOF</td>
<td></td>
</tr>
<tr>
<td>Connection</td>
<td>DAD, ODBC</td>
</tr>
<tr>
<td>EOF</td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>RecordCount</td>
<td></td>
</tr>
<tr>
<td>RecordStatus</td>
<td>DAD, ODBC</td>
</tr>
<tr>
<td>Restartable</td>
<td></td>
</tr>
<tr>
<td>StillExecuting</td>
<td>DAD, ODBC</td>
</tr>
<tr>
<td>Transactions</td>
<td>Always False</td>
</tr>
<tr>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>Updatable</td>
<td>False</td>
</tr>
<tr>
<td>UpdateOptions</td>
<td>DAD, ODBC</td>
</tr>
<tr>
<td>ValidationRule</td>
<td></td>
</tr>
</tbody>
</table>
ValidationText
Groups Collection

A **Groups** collection contains all stored **Group** objects of a **Workspace** or **user account** (Microsoft Jet workspaces only).

**Remarks**

You can append an existing **Group** object to the **Groups** collection in a **User** object to establish membership of a user account in that **Group** object. Alternatively, you can append a **User** object to the **Users** collection in a **Group** object.
object to give a user account the global permissions of that group. In either case, the existing Group object must already be a member of the Groups collection of the current Workspace object. If you use a Groups or Users collection other than the one to which you just appended an object, you may need to use the Refresh method to refresh the collection with current information from the database.

To refer to a Group object in a collection by its ordinal number or by its Name property setting, use any of the following syntax forms:

Groups(0)

Groups("name")

Groups![name]
Group Object

A **Group** object represents a group of **user accounts** that have common access **permissions** when a **Workspace** object operates as a **secure workgroup**. **(Microsoft Jet workspaces only).**

**Remarks**

You create **Group** objects and then use their names to establish and enforce access permissions for your databases, tables, and queries using the **Document**
objects that represent the Database, TableDef, and QueryDef objects with which you're working.

With the properties of a Group object, you can:

- Use the Name property of an existing Group object to return its name. You can't return the PID property setting of an existing Group object.

- Use the Name and PID properties of a newly created, unappended Group object to set the identity of that Group object.

You can append an existing Group object to the Groups collection in a User object to establish membership of a user account in that Group object. Alternatively, you can append a User object to the Users collection in a Group object to give a user account the global permissions of that group. If you use a Groups or Users collection other than the one to which you just appended an object, you may need to use the Refresh method to refresh the collection with current information from the database.

The Microsoft Jet database engine predefines three Group objects named Admins, Users, and Guests. To create a new Group object, use the CreateGroup method on a User or Workspace object.

To refer to a Group object in a collection by its ordinal number or by its Name property setting, use any of the following syntax forms:

Groups(0)

Groups("name")

Groups![name]
Group Object, Groups Collection

Summary

Group Object

A Group object contains these collections, this method, and these properties.

Collections

Properties

Users (default)

Method

CreateUser

Properties

Name

PID

Groups Collection

A Groups collection appears in each User and Microsoft Jet Workspace object, and contains these methods and this property.

Methods
Indexes Collection

An Indexes collection contains all the stored Index objects of a TableDef object (Microsoft Jet workspaces only).

Remarks

When you access a table-type Recordset object, use the object’s Index property to specify the order of records. Set this property to the Name property setting of an existing Index object in the Indexes collection of the the TableDef object underlying the Recordset object.
Note  You can use the **Append** or **Delete** method on an **Indexes** collection only if the **Updatable** property setting of the containing **TableDef** object is **True**.

After you create a new **Index** object, you should use the **Append** method to add it to the **TableDef** object's **Indexes** collection.

**Important**  Make sure your data complies with the attributes of your new index. If your index requires unique values, make sure that there are no duplicates in existing data records. If duplicates exist, the Microsoft Jet database engine can't create the index; a trappable error results when you attempt to use the **Append** method on the new index.
Index Object

Index objects specify the order of records accessed from database tables and whether or not duplicate records are accepted, providing efficient access to data. For external databases, Index objects describe the indexes established for external tables (Microsoft Jet workspaces only).

Remarks

The Microsoft Jet database engine uses indexes when it joins tables and creates Recordset objects. Indexes determine the order in which table-type Recordset
objects return records, but they don't determine the order in which the Microsoft Jet database engine stores records in the base table or the order in which any other type of **Recordset** object returns records.

With an **Index** object, you can:

- Use the **Required** property to determine whether the **Field** objects in the index require values that are not **Null**, and then use the **IgnoreNulls** property to determine whether the **Null** values have index entries.

- Use the **Primary** and **Unique** properties to determine the ordering and uniqueness of the **Index** object.

The Microsoft Jet database engine maintains all **base table** indexes automatically. It updates indexes whenever you add, change, or delete records from the base table. Once you create the database, use the **CompactDatabase** method periodically to bring index statistics up-to-date.

When accessing a table-type **Recordset** object, you specify the order of records using the object's **Index** property. Set this property to the **Name** property setting of an existing **Index** object in the **Indexes** collection. This collection is contained by the **TableDef** object underlying the **Recordset** object that you're populating.

**Note** You don't have to create indexes for a table, but for large, unindexed tables, accessing a specific record or processing **joins** can take a long time. Conversely, having too many indexes can slow down updates to the database as each of the table indexes is amended.

The **Attributes** property of each **Field** object in the index determines the order of records returned and consequently determines which access techniques to use for that index.

Each **Field** object in the **Fields** collection of an **Index** object is a component of the index. To define a new **Index** object, set its properties before you append it to a collection, making the **Index** object available for subsequent use.

**Note** You can modify the **Name** property setting of an existing **Index** object only if the **Updatable** property setting of the containing **TableDef** object is **True**.
When you set a primary key for a table, the Microsoft Jet database engine automatically defines it as the primary index. A primary index consists of one or more fields that uniquely identify all records in a table in a predefined order. Because the primary index field must be unique, the Microsoft Jet database engine automatically sets the Unique property of the primary Index object to True. If the primary index consists of more than one field, each field can contain duplicate values, but the combination of values from all the indexed fields must be unique. A primary index consists of a key for the table and is always made up of the same fields as the primary key.

**Important**  Make sure your data complies with the attributes of your new index. If your index requires unique values, make sure that there are no duplicates in existing data records. If duplicates exist, the Microsoft Jet database engine can't create the index; a trappable error results when you attempt to use the Append method on the new index.

When you create a relationship that enforces referential integrity, the Microsoft Jet database engine automatically creates an index with the Foreign property, set as the foreign key in the referencing table. After you've established a table relationship, the Microsoft Jet database engine prevents additions or changes to the database that violate that relationship. If you set the Attributes property of the Relation object to allow cascading updates and cascading deletes, the Microsoft Jet database engine updates or deletes records in related tables automatically.
To create a new Index object

1. Use the `CreateIndex` method on a `TableDef` object.

2. Use the `CreateField` method on the `Index` object to create a `Field` object for each field (column) to be included in the `Index` object.

3. Set `Index` properties as needed.

4. Append the `Field` object to the `Fields` collection.

5. Append the `Index` object to the `Indexes` collection.

   **Note** The `Clustered` property is ignored for databases that use the Microsoft Jet database engine, which doesn't support clustered indexes.
Index Object, Indexes Collection Summary

Index Object

An Index object contains these collections, methods, and properties.

Collections

Fields (default)
Properties

Methods

CreateField
CreateProperty

Properties

Clustered
DistinctCount
Foreign
IgnoreNulls
Name
Primary
Indexes Collection

An Indexes collection appears in each TableDef object, and contains these methods and this property.

Methods

Append
Delete
Refresh

Property

Count
A **Parameters** collection contains all the **Parameter** objects of a **QueryDef** object.

**Remarks**

The **Parameters** collection provides information only about existing parameters. You can't append objects to or delete objects from the **Parameters** collection.
Parameter Object

A Parameter object represents a value supplied to a query. The parameter is associated with a QueryDef object created from a parameter query.

Remarks

Parameter objects allow you to change the arguments in a frequently run QueryDef object without having to recompile the query.

Using the properties of a Parameter object, you can set a query parameter that can be changed before the query is run. You can:
• Use the **Name** property to return the name of a parameter.

• Use the **Value** property to set or return the parameter values to be used in the query.

• Use the **Type** property to return the data type of the **Parameter** object.

• Use the **Direction** property to set or return whether the parameter is an input parameter, an output parameter, or both.

In an **ODBCDirect workspace**, you can also:

• Change the setting of the **Type** property. Doing so will also clear the Value property.

• Use the **Direction** property to set or return whether the parameter is an input parameter, an output parameter, or both.
Parameter Object, Parameters Collection Summary

Parameter Object

A Parameter object contains no methods; it contains this collection and these properties.

Legend:

Feature available in ODBCDirect workspaces only.

Collection

Properties

Properties

Parameters Collection

A Parameters collection appears in each QueryDef object and contains this method and this property.
<table>
<thead>
<tr>
<th>Method</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refresh</td>
<td>Count</td>
</tr>
</tbody>
</table>
Properties Collection

A **Properties** collection contains all the **Property** objects for a specific instance of an object.

**Remarks**

Every DAO object except the **Connection** and **Error** objects contains a **Properties** collection, which has certain built-in **Property** objects. These **Property** objects (which are often just called properties) uniquely characterize that instance of the object.
In addition to the built-in properties, you can also create and add your own user-defined properties. To add a user-defined property to an existing instance of an object, first define its characteristics with the `CreateProperty` method, then add it to the collection with the `Append` method. Referencing a user-defined `Property` object that has not yet been appended to a `Properties` collection will cause an error, as will appending a user-defined `Property` object to a `Properties` collection containing a `Property` object of the same name.

You can use the `Delete` method to remove user-defined properties from the `Properties` collection, but you can't remove built-in properties.

**Note** A user-defined `Property` object is associated only with the specific instance of an object. The property isn't defined for all instances of objects of the selected type.

You can use the `Properties` collection of an object to enumerate the object's built-in and user-defined properties. You don't need to know beforehand exactly which properties exist or what their characteristics (Name and Type properties) are to manipulate them. However, if you try to read a write-only property, such as the Password property of a `Workspace` object, or try to read or write a property in an inappropriate context, such as the Value property setting of a `Field` object in the `Fields` collection of a `TableDef` object, an error occurs.

To refer to a built-in `Property` object in a collection by its ordinal number or by its Name property setting, use any of the following syntax forms:

```plaintext
object.Properties(0)
object.Properties("name")
object.Properties![name]
```

For a built-in property, you can also use this syntax:

```plaintext
object.name
```

**Note** For a user-defined property, you must use the full `object.Properties("name")` syntax.

With the same syntax forms, you can also refer to the Value property of a
**Property** object. The context of the reference will determine whether you are referring to the **Property** object itself or the **Value** property of the **Property** object.
Property Object

A Property object represents a built-in or user-defined characteristic of a DAO object.

Remarks

Every DAO object except the Connection and Error objects contains a Properties collection which has Property objects corresponding to built-in properties of that DAO object. The user can also define Property objects and append them to the Properties collection of some DAO objects. These Property objects (which are often just called properties) uniquely characterize that
instance of the object.

You can create user-defined properties for the following objects:

- **Database, Index, QueryDef, and TableDef objects**
- **Field objects in Fields collections of QueryDef and TableDef objects**

To add a user-defined property, use the `CreateProperty` method to create a `Property` object with a unique `Name` property setting. Set the `Type` and `Value` properties of the new `Property` object, and then append it to the `Properties` collection of the appropriate object. The object to which you are adding the user-defined property must already be appended to a collection. Referencing a user-defined `Property` object that has not yet been appended to a `Properties` collection will cause an error, as will appending a user-defined `Property` object to a `Properties` collection containing a `Property` object of the same name.

You can delete user-defined properties from the `Properties` collection, but you can't delete built-in properties.

**Note**  A user-defined `Property` object is associated only with the specific instance of an object. The property isn't defined for all instances of objects of the selected type.

You can use the `Properties` collection of an object to enumerate the object's built-in and user-defined properties. You don't need to know beforehand exactly which properties exist or what their characteristics (Name and Type properties) are to manipulate them. However, if you try to read a write-only property, such as the `Password` property of a `Workspace` object, or try to read or write a property in an inappropriate context, such as the `Value` property setting of a `Field` object in the `Fields` collection of a `TableDef` object, an error occurs.

The `Property` object also has four built-in properties:

- The **Name** property, a `String` that uniquely identifies the property.
- The **Type** property, an `Integer` that specifies the property data type.
- The **Value** property, a `Variant` that contains the property setting.
• The **Inherited** property, a **Boolean** that indicates whether the property is inherited from another object. For example, a **Field** object in a **Fields** collection of a **Recordset** object can inherit properties from the underlying **TableDef** or **QueryDef** object.

To refer to a built-in **Property** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

```
object.Properties(0)
```

```
object.Properties("name")
```

```
object.Properties![name]
```

For a built-in property, you can also use this syntax:

```
object.name
```

**Note** For a user-defined property, you must use the full `object.Properties("name")` syntax.

With the same syntax forms, you can also refer to the **Value** property of a **Property** object. The context of the reference will determine whether you are referring to the **Property** object itself or the **Value** property of the **Property** object.
Property Object, Properties Collection Summary

Property Object

A Property object contains no methods; it contains this collection and these properties.

Collection

Properties

Properties

Inherited (Always False in ODBCDirect databases)

Name
Type
Value

Properties Collection

A Properties collection appears in each of the other DAO objects except the Connection and Error objects, and contains these methods and this property.

Methods

Append
Delete
QueryDefs Collection

A **QueryDefs** collection contains all **QueryDef** objects of a **Database** object in a **Microsoft Jet database**, and all **QueryDef** objects of a **Connection** object in an **ODBCDirect workspace**.

**Remarks**

To create a new **QueryDef** object, use the **CreateQueryDef** method. In a
Microsoft Jet workspace, if you supply a string for the name argument or if you explicitly set the Name property of the new QueryDef object to a non-zero-length string, you will create a permanent QueryDef that will automatically be appended to the QueryDefs collection and saved to disk. Supplying a zero-length string as the name argument or explicitly setting the Name property to a zero-length string will result in a temporary QueryDef object.

In an ODBCDirect workspace, a QueryDef is always temporary. The QueryDefs collection contains all open QueryDef objects. When a QueryDef is closed, it is automatically removed from the QueryDefs collection.

To refer to a QueryDef object in a collection by its ordinal number or by its Name property setting, use any of the following syntax forms:

**QueryDefs(0)**

**QueryDefs("name")**

**QueryDefs![name]**

You can refer to temporary QueryDef objects only by the object variables that you have assigned to them.
A **QueryDef** object is a stored definition of a query in a **Microsoft Jet** database, or a temporary definition of a query in an **ODBCDirect workspace**.

**Remarks**

You can use the **QueryDef** object to define a query. For example, you can:
• Use the SQL property to set or return the query definition.

• Use the QueryDef object's Parameters collection to set or return query parameters.

• Use the Type property to return a value indicating whether the query selects records from an existing table, makes a new table, inserts records from one table into another table, deletes records, or updates records.

• Use the MaxRecords property to limit the number of records returned from a query.

• Use the ODBCTimeout property to indicate how long to wait before the query returns records. The ODBCTimeout property applies to any query that accesses ODBC data.

In a Microsoft Jet workspace, you can also:

• Use the ReturnsRecords property to indicate that the query returns records. The ReturnsRecords property is only valid on SQL pass-through queries.

• Use the Connect property to make an SQL pass-through query to an ODC database.

In an ODBCDirect workspace, you can also:

• Use the Prepare property to determine whether to invoke the ODBC SQLPrepare API when the query is executed.

• Use the CacheSize property to cache records returned from a query.

You can also create temporary QueryDef objects. Unlike permanent QueryDef objects, temporary QueryDef objects are not saved to disk or appended to the QueryDefs collection. Temporary QueryDef objects are useful for queries that you must run repeatedly during run time but do not not need to save to disk, particularly if you create their SQL statements during run time.

You can think of a permanent QueryDef object in a Microsoft Jet workspaces as
a compiled SQL statement. If you execute a query from a permanent QueryDef object, the query will run faster than if you run the equivalent SQL statement from the OpenRecordset method. This is because the Microsoft Jet database engine doesn't need to compile the query before executing it.

The preferred way to use the native SQL dialect of an external database engine accessed through the Microsoft Jet database engine is through QueryDef objects. For example, you can create a Microsoft SQL Server query and store it in a QueryDef object. When you need to use a non-Microsoft Jet database engine SQL query, you must provide a Connect property string that points to the external data source. Queries with valid Connect properties bypass the Microsoft Jet database engine and pass the query directly to the external database server for processing.

To create a new QueryDef object, use the CreateQueryDef method. In a Microsoft Jet workspace, if you supply a string for the name argument or if you explicitly set the Name property of the new QueryDef object to a non–zero-length string, you will create a permanent QueryDef that will automatically be appended to the QueryDefs collection and saved to disk. Supplying a zero-length string as the name argument or explicitly setting the Name property to a zero-length string will result in a temporary QueryDef object.

In an ODBCDirect workspace, a QueryDef is always temporary. The QueryDefs collection contains all open QueryDef objects. When a QueryDef is closed, it is automatically removed from the QueryDefs collection.

To refer to a QueryDef object in a collection by its ordinal number or by its Name property setting, use any of the following syntax forms:

```
QueryDefs(0)
QueryDefs("name")
QueryDefs![name]
```

You can refer to temporary QueryDef objects only by the object variables that you have assigned to them.
QueryDef Object, QueryDefs Collection Summary

QueryDef Object

A [QueryDef](#) object contains these collections, methods, and properties.

Legend:

- Available only in a [Microsoft Jet workspace](#).
- Available only in an [ODBCDirect workspace](#).

Collections

- [Fields](#)
- [Parameters](#) (default)
- [Properties](#)

Methods

- [Cancel](#)
- [Close](#)
- [CreateProperty](#)
- [Execute](#)
- [OpenRecordset](#)
Properties

CacheSize  Connect  DateCreated  KeepLocal  LastUpdated  LogMessages  MaxRecords  Name  ODBCTimeout  Prepare  RecordsAffected  Replicable  ReturnsRecords  SQL  StillExecuting  Type  Updatable

QueryDefs Collection

A QueryDefs collection appears in each Connection object in an ODBCDirect workspace, and each Database object, and contains these methods and this property.

Methods

Append  Delete  Refresh

Property
Count
A **Recordsets** collection contains all open **Recordset** objects in a **Connection** or **Database** object.

**Remarks**

When you use DAO objects, you manipulate data almost entirely using **Recordset** objects.
A new **Recordset** object is automatically added to the **Recordsets** collection when you open the **Recordset** object, and is automatically removed when you close it.

You can create as many **Recordset** object variables as needed. Different **Recordset** objects can access the same tables, queries, and fields without conflicting.

To refer to a **Recordset** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

**Recordsets(0)**

**Recordsets(“name”)**

**Recordsets![name]**

**Note**  You can open a **Recordset** object from the same data source or database more than once, creating duplicate names in the **Recordsets** collection. You should assign **Recordset** objects to **object variables** and refer to them by variable name.
Recordset Object

A Recordset object represents the records in a base table or the records that result from running a query.

Remarks

You use Recordset objects to manipulate data in a database at the record level. When you use DAO objects, you manipulate data almost entirely using Recordset objects. All Recordset objects are constructed using records (rows)
and fields (columns). There are five types of Recordset objects:

- **Table-type Recordset** — representation in code of a base table that you can use to add, change, or delete records from a single database table (Microsoft Jet workspaces only).

- **Dynaset-type Recordset** — the result of a query that can have updatable records. A dynaset-type Recordset object is a dynamic set of records that you can use to add, change, or delete records from an underlying database table or tables. A dynaset-type Recordset object can contain fields from one or more tables in a database. This type corresponds to an ODBC keyset cursor.

- **Snapshot-type Recordset** — a static copy of a set of records that you can use to find data or generate reports. A snapshot-type Recordset object can contain fields from one or more tables in a database but can't be updated. This type corresponds to an ODBC static cursor.

- **Forward-only-type Recordset** — identical to a snapshot except that no cursor is provided. You can only scroll forward through records. This improves performance in situations where you only need to make a single pass through a result set. This type corresponds to an ODBC forward-only cursor.

- **Dynamic-type Recordset** — a query result set from one or more base tables in which you can add, change, or delete records from a row-returning query. Further, records other users add, delete, or edit in the base tables also appear in your Recordset. This type corresponds to an ODBC dynamic cursor (ODBCDirect workspaces only).

You can choose the type of Recordset object you want to create using the type argument of the OpenRecordset method.

In a Microsoft Jet workspace, if you don't specify a type, DAO attempts to create the type of Recordset with the most functionality available, starting with table. If this type isn’t available, DAO attempts a dynaset, then a snapshot, and finally a forward-only type Recordset object.

In an ODBCDirect workspace, if you don't specify a type, DAO attempts to
create the type of **Recordset** with the fastest query response, starting with forward-only. If this type isn't available, DAO attempts a snapshot, then a dynaset, and finally a dynamic- type **Recordset** object.

When creating a **Recordset** object using a non-linked **TableDef** object in a Microsoft Jet workspace, table-type **Recordset** objects are created. Only dynaset-type or snapshot-type **Recordset** objects can be created with linked tables or tables in Microsoft Jet-connected ODBC databases.

A new **Recordset** object is automatically added to the **Recordsets** collection when you open the object, and is automatically removed when you close it.

**Note**  If you use variables to represent a **Recordset** object and the **Database** object that contains the **Recordset**, make sure the variables have the same scope, or lifetime. For example, if you declare a public variable that represents a **Recordset** object, make sure the variable that represents the **Database** containing the **Recordset** is also public, or is declared in a **Sub** or **Function** procedure using the **Static** keyword.

You can create as many **Recordset** object variables as needed. Different **Recordset** objects can access the same tables, queries, and fields without conflicting.

Dynaset-, snapshot-, and forward-only–type **Recordset** objects are stored in local memory. If there isn't enough space in local memory to store the data, the Microsoft Jet database engine saves the additional data to **TEMP** disk space. If this space is exhausted, a trappable error occurs.

The default collection of a **Recordset** object is the **Fields** collection, and the default property of a **Field** object is the **Value** property. Use these defaults to simplify your code.

When you create a **Recordset** object, the current record is positioned to the first record if there are any records. If there are no records, the **RecordCount** property setting is 0, and the **BOF** and **EOF** property settings are **True**.

You can use the **MoveNext**, **MovePrevious**, **MoveFirst**, and **MoveLast** methods to reposition the current record. Forward-only–type **Recordset** objects support only the **MoveNext** method. When using the Move methods to visit each record (or "walk" through the **Recordset**), you can use the **BOF** and **EOF**
properties to check for the beginning or end of the **Recordset** object.

With dynaset- and snapshot-type **Recordset** objects in a Microsoft Jet workspace, you can also use the Find methods, such as **FindFirst**, to locate a specific record based on criteria. If the record isn't found, the **NoMatch** property is set to **True**. For table-type **Recordset** objects, you can scan records using the **Seek** method.

The **Type** property indicates the type of **Recordset** object created, and the **Updatable** property indicates whether you can change the object's records.

Information about the structure of a base table, such as the names and data types of each **Field** object and any **Index** objects, is stored in a **TableDef** object.

To refer to a **Recordset** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

**Recordsets**(0)

**Recordsets**("name")

**Recordsets**![name]

**Note**  You can open a **Recordset** object from the same data source or database more than once, creating duplicate names in the **Recordsets** collection. You should assign **Recordset** objects to **object variables** and refer to them by variable name.
Recordset Object, Recordsets
Collection Summary

Recordset Object

A Recordset object contains these collections, methods, and properties.

Legend:

✔️  Feature available in both Microsoft Jet and ODBCDirect workspaces.

✍️  Feature available in Microsoft Jet workspaces only.

بيانات ODBC  Feature available in ODBCDirect workspaces only.

Collections

Fields (default)
Properties

Recordset Methods

The following table lists all of the Recordset methods, and shows which Recordset type supports each method, and whether the method is available in either a Microsoft Jet or ODBCDirect workspace, or both.

<table>
<thead>
<tr>
<th>Method</th>
<th>Table</th>
<th>Dynaset</th>
<th>Snapshot</th>
<th>Forward-Only</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>DAOD</td>
<td>ODBC</td>
<td>*</td>
<td>DAOD</td>
<td>ODBC</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------</td>
<td>-------</td>
<td>---</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>AddNew</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td>DAOD</td>
<td>ODBC</td>
<td></td>
<td>DAOD</td>
<td>ODBC</td>
</tr>
<tr>
<td>CancelUpdate</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clone</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CopyQueryDef</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edit</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FillCache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FindFirst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FindLast</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FindNext</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FindPrevious</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetRows</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* In an ODBCDirect workspace, a snapshot-type **Recordset** may be updatable, depending on the ODBC driver. The **AddNew**, **Edit**, **Delete**, **Update**, and **CancelUpdate** methods are only available on ODBCDirect snapshot-type **Recordset** objects if the ODBC driver supports updatable snapshots.
Recordset Properties

The following table indicates which properties apply to each type of Recordset object and whether the property setting is read/write, read-only, or always False in either Microsoft Jet or ODBCDirect databases.

- **Read-only**
- **Read/write**

<table>
<thead>
<tr>
<th>Property</th>
<th>Table</th>
<th>Dynaset</th>
<th>Snapshot</th>
<th>Forward-Only</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsolutePosition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BatchCollisionCount</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BatchCollisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BatchSize</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bookmark</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bookmarkable</td>
<td>for</td>
<td>for</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CacheSize</td>
<td>for</td>
<td></td>
<td>for</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CacheStart</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DateCreated</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EditMode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LastModified</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LastUpdated</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* for Microsoft Jet workspaces
  
  for ODBCDirect workspaces
<table>
<thead>
<tr>
<th>Field</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td>Microsoft Jet workspaces for ODBCDirect workspaces</td>
</tr>
<tr>
<td>NoMatch</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>PercentPosition</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>RecordCount</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>RecordStatus</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>Restartable</td>
<td>False</td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>Sort</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>StillExecuting</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>Transactions</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>Updatable</td>
<td>False</td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>UpdateOptions</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>ValidationRule</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
<tr>
<td>ValidationText</td>
<td></td>
<td><strong>DAO</strong>, <strong>ODBC</strong></td>
</tr>
</tbody>
</table>

*In an ODBCDirect workspace, a snapshot-type Recordset may be updatable, depending on the ODBC driver. The LastModified property is available, and the
**Updatable** property is **True** only on ODBCDirect snapshot-type **Recordset** objects if the ODBC driver supports updatable snapshots.

**Recordsets Collection**

A **Recordsets** collection appears in each **Connection** and **Database** object, and contains this method and this property.

**Method**

**Refresh**

**Property**

**Count**
Relations Collection

A Relations collection contains stored Relation objects of a Database object (Microsoft Jet databases only).

Remarks

You can use the Relation object to create new relationships and examine existing relationships in your database. To add a Relation object to the Relations collection, first create it with the CreateRelation method, and then append it to the Relations collection with the Append method. This will save
the **Relation** object when you close the **Database** object. To remove a **Relation** object from the collection, use the **Delete** method.

To refer to a **Relation** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

- **Relations**(0)
- **Relations**("name")
- **Relations**![name]
A **Relation** object represents a **relationship** between fields in tables or queries (Microsoft Jet databases only).

**Remarks**

You can use the **Relation** object to create new relationships and examine existing relationships in your database.

Using a **Relation** object and its properties, you can:
- Specify an enforced relationship between fields in base tables (but not a relationship that involves a query or a linked table).

- Establish unenforced relationships between any type of table or query — native or linked.

- Use the Name property to refer to the relationship between the fields in the referenced primary table and the referencing foreign table.

- Use the Attributes property to determine whether the relationship between fields in the table is one-to-one or one-to-many and how to enforce referential integrity.

- Use the Attributes property to determine whether the Microsoft Jet database engine can perform cascading update and cascading delete operations on primary and foreign tables.

- Use the Attributes property to determine whether the relationship between fields in the table is left join or right join.

- Use the Name property of all Field objects in the Fields collection of a Relation object to set or return the names of the fields in the primary key of the referenced table, or the ForeignName property settings of the Field objects to set or return the names of the fields in the foreign key of the referencing table.

If you make changes that violate the relationships established for the database, a trappable error occurs. If you request cascading update or cascading delete operations, the Microsoft Jet database engine also modifies the primary or foreign key tables to enforce the relationships you establish.

For example, the Northwind database contains a relationship between an Orders table and a Customers table. The CustomerID field of the Customers table is the primary key, and the CustomerID field of the Orders table is the foreign key. For Microsoft Jet to accept a new record in the Orders table, it searches the Customers table for a match on the CustomerID field of the Orders table. If Microsoft Jet doesn't find a match, it doesn't accept the new record, and a trappable error occurs.
When you enforce referential integrity, a unique index must already exist for the key field of the referenced table. The Microsoft Jet database engine automatically creates an index with the **Foreign** property set to act as the foreign key in the referencing table.

To create a new **Relation** object, use the **CreateRelation** method. To refer to a **Relation** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

```
Relations(0)

Relations("name")

Relations![name]
```
Relation Object, Relations Collection

Summary

Relation Object

A Relation object contains these collections, this method, and these properties.

Collections

Fields (Default)
Properties

Method

CreateField

Properties

Attributes
ForeignTable
Name
PartialReplica
Table

Relations Collection
A **Relations** collection is contained in each **Database** object of a **Microsoft Jet database**, and contains these methods and this property.

**Methods**

- Append
- Delete
- Refresh

**Property**

- Count
A snapshot-type Recordset object is a static set of records that you can use to examine data in an underlying table or tables. In an ODBCDirect database, a snapshot-type Recordset object corresponds to a static cursor.

Remarks

To create a snapshot-type Recordset object, use the OpenRecordset method on an open database, on another dynaset- or snapshot-type Recordset object, or on a QueryDef object.

A snapshot-type Recordset object can contain fields from one or more tables in a database. In a Microsoft Jet workspace, a snapshot can't be updated. In an ODBCDirect workspace, a snapshot may be updatable, depending on the ODBC driver.
When you create a snapshot-type **Recordset** object, data values for all fields (except **Memo** and **OLE Object** (Long Binary) field data types in .mdb files) are brought into memory. Once loaded, changes made to **base table** data aren't reflected in the snapshot-type **Recordset** object data. To reload the snapshot-type **Recordset** object with current data, use the **Requery** method, or re-execute the **OpenRecordset** method.

The order of snapshot-type **Recordset** object data doesn't necessarily follow any specific sequence. To order your data, use an **SQL statement** with an **ORDER BY** clause to create the **Recordset** object. You can also use this technique to filter the records so that only certain records are added to the **Recordset** object. Using this technique instead of using the **Filter** or **Sort** properties or testing each record individually generally results in faster access to your data.

Snapshot-type **Recordset** objects are generally faster to create and access than dynaset-type **Recordset** objects because their records are either in memory or stored in **TEMP** disk space, and the **Microsoft Jet database engine** doesn't need to lock pages or handle multiuser issues. However, snapshot-type **Recordset** objects use more resources than **dynaset-type Recordset** objects because the entire record is downloaded to local memory.
The snapshot-type Recordset object contains these collections, methods, and properties.

Legend:

- Feature available in **Microsoft Jet workspaces** only.
- Feature available in **ODBCDirect workspaces** only.

### Collections

- **Fields** (default)
- **Properties**

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddNew</td>
<td>DAO, ODBC *</td>
</tr>
<tr>
<td>Cancel</td>
<td>DAO, ODBC</td>
</tr>
<tr>
<td>CancelUpdate</td>
<td>DAO, ODBC *</td>
</tr>
<tr>
<td>Clone</td>
<td>DAO</td>
</tr>
<tr>
<td>Close</td>
<td></td>
</tr>
<tr>
<td>CopyQueryDef</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>DAO, ODBC *</td>
</tr>
<tr>
<td>Edit</td>
<td>DAO, ODBC *</td>
</tr>
</tbody>
</table>
FindFirst
FindLast
FindNext
FindPrevious
GetRows
Move
MoveFirst
MoveLast
MoveNext
MovePrevious
NextRecordset
OpenRecordset
Requery
Update

* In an ODBCDirect workspace, a snapshot-type Recordset may be updatable, depending on the ODBC driver. The AddNew, Edit, Delete, Update, and CancelUpdate methods are only available on ODBCDirect snapshot-type Recordset objects if the ODBC driver supports updatable snapshots.

Properties

The following table indicates whether the property setting is read/write, read-only, or always False in either Microsoft Jet or ODBCDirect workspaces.

![Read-only]

Read-only

![Read/write]

Read/write

<table>
<thead>
<tr>
<th>Properties</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsolutePosition</td>
<td><img src="Read-only" alt="" /></td>
</tr>
<tr>
<td>BatchCollisionCount</td>
<td><img src="Read-only" alt="" /></td>
</tr>
<tr>
<td>BatchCollisions</td>
<td><img src="Read-only" alt="" /></td>
</tr>
<tr>
<td>BatchSize</td>
<td><img src="Read-only" alt="" /></td>
</tr>
<tr>
<td>BOF</td>
<td><img src="Read-only" alt="" /></td>
</tr>
</tbody>
</table>
* In an ODBCDirect workspace, a snapshot-type Recordset may be updatable, depending on the ODBC driver. The LastModified property is available, and the Updatable property is True only on ODBCDirect snapshot-type Recordset objects if the ODBC driver supports updatable snapshots.
TableDefs Collection

A TableDefs collection contains all stored TableDef objects in a database (Microsoft Jet workspaces only).

Remarks

You manipulate a table definition using a TableDef object and its methods and properties.
The default collection of a `Database` object is the `TableDefs` collection.

To refer to a `TableDef` object in a collection by its ordinal number or by its `Name` property setting, use any of the following syntax forms:

`TableDefs(0)`

`TableDefs("name")`

`TableDefs![name]`
TableDef Object

A TableDef object represents the stored definition of a base table or a linked table (Microsoft Jet workspaces only).

Remarks

You manipulate a table definition using a TableDef object and its methods and properties. For example, you can:
Examine the field and index structure of any local, linked, or external table in a database.

Use the Connect and SourceTableName properties to set or return information about linked tables, and use the RefreshLink method to update connections to linked tables.

Use the ValidationRule and ValidationText properties to set or return validation conditions.

Use the OpenRecordset method to create a table-, dynaset-, dynamic-, snapshot-, or forward-only–type Recordset object, based on the table definition.

For base tables, the RecordCount property contains the number of records in the specified database table. For linked tables, the RecordCount property setting is always -1.

To create a new TableDef object, use the CreateTableDef method.
To add a field to a table

1. Make sure any Recordset objects based on the table are all closed.

2. Use the CreateField method to create a Field object variable and set its properties.

3. Use the Append method to add the Field object to the Fields collection of the TableDef object.

You can delete a Field object from a TableDefs collection if it doesn't have any indexes assigned to it, but you will lose the field's data.
To create a table that is ready for new records in a database

1. Use the **CreateTableDef** method to create a **TableDef** object.

2. Set its properties.

3. For each field in the table, use the **CreateField** method to create a **Field** object variable and set its properties.

4. Use the **Append** method to add the fields to the **Fields** collection of the **TableDef** object.

5. Use the **Append** method to add the new **TableDef** object to the **TableDefs** collection of the **Database** object.

A linked table is connected to the database by the **SourceTableName** and **Connect** properties of the **TableDef** object.
To link a table to a database

1. Use the `CreateTableDef` method to create a `TableDef` object.

2. Set its `Connect` and `SourceTableName` properties (and optionally, its `Attributes` property).

3. Use the `Append` method to add it to the `TableDefs` collection of a `Database`.

To refer to a `TableDef` object in a collection by its ordinal number or by its `Name` property setting, use any of the following syntax forms:

```csharp
TableDefs(0)
TableDefs("name")
TableDefs![name]
```
TableDef Object, TableDefs Collection Summary

TableDef Object

A TableDef object contains these collections, methods, and properties.

Collections

- Fields (Default)
- Indexes
- Properties

Methods

- CreateField
- CreateIndex
- CreateProperty
- OpenRecordset
- RefreshLink

Properties

- Attributes
- ConflictTable
A **TableDef** object may also contain application-defined properties. For details on reading and setting these properties, refer to the application's online Help.

**TableDefs Collection**

A **TableDefs** collection is contained in each **Database** object in a Microsoft Jet database, and contains these methods and this property.

**Methods**

- Append
- Delete
- Refresh

**Property**

- Count
Table-Type Recordset Object

A table-type **Recordset** object represents a **base table** you can use to add, change, or delete records from a table. Only the **current record** is loaded into memory. A predefined **index** determines the order of the records in the **Recordset** object (**Microsoft Jet workspaces** only).

**Remarks**

To create a table-type **Recordset** object, use the **OpenRecordset** method on an open **Database** object.

You can create a table-type **Recordset** object from a base table of a **Microsoft Jet database**, but not from an **ODBC** or **linked table**. You can use the table-type **Recordset** object with ISAM databases (like Excel, dBASE, or Paradox) when you open them directly.
Unlike dynaset- or snapshot-type Recordset objects, the table-type Recordset object can't refer to more than one base table, and you can't create it with an SQL statement that filters or sorts the data. Generally, when you access a table-type Recordset object, you specify one of the predefined indexes for the table, which orders the data returned to your application. If the table doesn't have an index, the data won't necessarily be in a particular order. If necessary, your application can create an index that returns records in a specific order. To choose a specific order for your table-type Recordset object, set the Index property to a valid index.

Also unlike dynaset- or snapshot-type Recordset objects, you don't need to explicitly populate table-type Recordset objects to obtain an accurate value for the RecordCount property.

To maintain data integrity, table-type Recordset objects are locked during the Edit and Update methods operations so that only one user can update a particular record at a time. When the Microsoft Jet database engine locks a record, it locks the entire 2K page containing the record.

Two kinds of locking are used with non-ODBC tables — pessimistic and optimistic. ODBC-accessed tables always use optimistic locking. The LockEdits property determines the locking conditions in effect during editing.
Table-Type Recordset Object Summary

A **table-type Recordset** object contains these collections, methods, and properties. This type of **Recordset** and its methods and properties are available only in a **Microsoft Jet workspace**.

**Collections**

- **Fields** (default)
- **Properties**

**Methods**

- **AddNew**
- **CancelUpdate**
- **Clone**
- **Close**
- **Delete**
- **Edit**
- **GetRows**
- **Move**
- **MoveFirst**
- **MoveLast**
- **MoveNext**
- **MovePrevious**
Properties

The following table indicates whether each property setting is read/write, read-only, or always **False**.

- **Read-only**
- **Read/write**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOF</td>
<td></td>
</tr>
<tr>
<td>Bookmark</td>
<td></td>
</tr>
<tr>
<td>Bookmarkable</td>
<td></td>
</tr>
<tr>
<td>DateCreated</td>
<td></td>
</tr>
<tr>
<td>EditMode</td>
<td></td>
</tr>
<tr>
<td>EOF</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td></td>
</tr>
<tr>
<td>LastModified</td>
<td></td>
</tr>
<tr>
<td>LastUpdated</td>
<td></td>
</tr>
<tr>
<td>LockEdits</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>NoMatch</td>
<td></td>
</tr>
<tr>
<td>PercentPosition</td>
<td></td>
</tr>
<tr>
<td>RecordCount</td>
<td></td>
</tr>
<tr>
<td>Restartable</td>
<td><strong>Always False</strong></td>
</tr>
<tr>
<td>Transactions</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>Updatable</td>
<td></td>
</tr>
<tr>
<td>ValidationRule</td>
<td></td>
</tr>
<tr>
<td>ValidationText</td>
<td></td>
</tr>
</tbody>
</table>
Users Collection

A Users collection contains all stored User objects of a Workspace or Group object (Microsoft Jet workspaces only).

Remarks

You can append an existing User object to the Users collection of a Group object to give a user account the access permissions for that Group object. Alternatively, you can append the Group object to the Groups collection in a
**User** object to establish membership of the user account in that group. If you use a **Users** or **Groups** collection other than the one to which you just appended an object, you may need to use the **Refresh** method.

The **Microsoft Jet database engine** predefines two **User** objects named Admin and Guest. The user Admin is a member of both of the **Group** objects named Admins and Users; the user Guest is a member only of the **Group** object named Guests.

To refer to a **User** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

```
[workspace | group].Users(0)

[workspace | group].Users("name")

[workspace | group].Users![name]
```
A User object represents a user account that has access permissions when a Workspace object operates as a secure workgroup (Microsoft Jet workspaces only).

Remarks

You use User objects to establish and enforce access permissions for the Document objects that represent databases, tables, and queries. Also, if you
know the properties of a specific User object, you can create a new Workspace object that has the same access permissions as the User object.

You can append an existing User object to the Users collection of a Group object to give a user account the access permissions for that Group object. Alternatively, you can append the Group object to the Groups collection in a User object to establish membership of the user account in that group. If you use a Users or Groups collection other than the one to which you just appended an object, you may need to use the Refresh method.

With the properties of a User object, you can:

- Use the Name property to return the name of an existing user. You can't return the PID and Password properties of an existing User object.

- Use the Name, PID, and Password properties of a newly created, unappended User object to establish the identity of that User object. If you don't set the Password property, it's set to a zero-length string ("").

The Microsoft Jet database engine predefines two User objects named Admin and Guest. The user Admin is a member of both of the Group objects named Admins and Users; the user Guest is a member only of the Group object named Guests.

To create a new User object, use the CreateUser method.

To refer to a User object in a collection by its ordinal number or by its Name property setting, use any of the following syntax forms:

```
[workspace | group].Users(0)
[workspace | group].Users("name")
[workspace | group].Users![name]
```
User Object, Users Collection

Summary

User Object

A User object contains these collections, methods, and properties.

Collections

Groups (Default)
Properties

Methods

CreateGroup
NewPassword

Properties

Name
Password
PID

Users Collection
A Users collection is contained in each Group and Microsoft Jet Workspace object, and contains these methods and this property.

**Methods**

- Append
- Delete
- Refresh

**Property**

- Count
A **Workspaces** collection contains all active, unhidden **Workspace** objects of the **DBEngine** object. (Hidden **Workspace** objects are not appended to the collection and referenced by the variable to which they are assigned.)
Use the **Workspace** object to manage the current session or to start an additional session.

When you first refer to or use a **Workspace** object, you automatically create the default workspace, `DBEngine.Workspaces(0)`. The settings of the **Name** and **UserName** properties of the default workspace are 
"#Default Workspace#" and "Admin," respectively. If security is enabled, the **UserName** property setting is the name of the user who logged on.

You can create new **Workspace** objects with the **CreateWorkspace** method. After you create a new **Workspace** object, you must append it to the **Workspaces** collection if you need to refer to it from the **Workspaces** collection. You can, however, use a newly created **Workspace** object without appending it to the **Workspaces** collection.

To refer to a **Workspace** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

```csharp
DBEngine.Workspaces(0)
```

```csharp
DBEngine.Workspaces("name")
```

```csharp
DBEngine.Workspaces![name]
```
A *Workspace* object defines a named *session* for a user. It contains open databases and provides mechanisms for simultaneous *transactions* and, in *Microsoft Jet* workspaces, *secure workgroup* support. It also controls whether you are going through the *Microsoft Jet database engine* or *ODBCDirect* to access external data.
Remarks

A **Workspace** is a non-**persistent object** that defines how your application interacts with data — either by using the Microsoft Jet database engine, or **ODBCDirect**. Use the **Workspace** object to manage the current session or to start an additional session. In a session, you can open multiple databases or connections, and manage transactions. For example, you can:

- Use the **Name**, **UserName**, and **Type** properties to establish a named session. The session creates a scope in which you can open multiple databases and conduct one instance of nested transactions.

- Use the **Close** method to terminate a session.

- Use the **OpenDatabase** method to open one or more existing databases on a **Workspace**.

- Use the **BeginTrans**, **CommitTrans**, and **Rollback** methods to manage nested transaction processing within a **Workspace** and use several **Workspace** objects to conduct multiple, simultaneous, and overlapping transactions.

Further, using a Microsoft Jet database, you can establish security based on user names and passwords:

- Use the **Groups** and **Users** collections to establish group and user access permissions to objects in the **Workspace**.

- Use the **IsolateODBCTrans** property to isolate multiple transactions that involve the same Microsoft Jet-connected ODBC database.

**Note**  For a complete list of all methods, properties, and collections available on a **Workspace** object in either a Microsoft Jet database or an ODBCDirect database, see the **Summary** topic.

When you first refer to or use a **Workspace** object, you automatically create the **default workspace**, `DBEngine.Workspaces(0)`. The settings of the **Name** and **UserName** properties of the default workspace are ":Default Workspace#" and "Admin," respectively. If security is enabled, the **UserName** property setting is
the name of the user who logged on.

To establish an ODBCDirect **Workspace** object, and thereby avoid loading the Microsoft Jet database engine into memory, set the **DBEngine** object's **DefaultType** property to **dbUseODBC**, or set the **type** argument of the **CreateWorkspace** method to **dbUseODBC**.

When you use transactions, all databases in the specified **Workspace** are affected — even if multiple **Database** objects are opened in the **Workspace**. For example, you use a **BeginTrans** method, update several records in a database, and then delete records in another database. If you then use the **Rollback** method, both the update and delete operations are canceled and rolled back. You can create additional **Workspace** objects to manage transactions independently across **Database** objects.

You can create **Workspace** objects with the **CreateWorkspace** method. After you create a new **Workspace** object, you must append it to the **Workspaces** collection if you need to refer to it from the **Workspaces** collection.

You can use a newly created **Workspace** object without appending it to the **Workspaces** collection. However, you must refer to it by the **object variable** to which you have assigned it.

To refer to a **Workspace** object in a collection by its ordinal number or by its **Name** property setting, use any of the following syntax forms:

```plaintext
DBEngine.Workspaces(0)

DBEngine.Workspaces("name")

DBEngine.Workspaces![name]
```
Workspace Object, Workspaces
Collection Summary

Workspace Object

A **Workspace** object contains these collections, methods, and properties.

Legend:

- Feature available in [Microsoft Jet workspaces](#) only.
- Feature available in [ODBCDirect workspaces](#) only.

Collections

- **Connections**
- **Databases** (default)
- **Groups**
- **Properties**
- **Users**

Methods

- **BeginTrans**
- **Close**
- **CommitTrans**
- **CreateDatabase**
Properties

- **DefaultCursorDriver**
- **IsolateODBCTrans**
- **LoginTimeout**
- **Name**
- **Type**
- **UserName**

Workspaces Collection

A Workspaces collection is contained in the DBEngine object, and contains these methods and this property.

Methods

- **Append**
- **Delete**
- **Refresh**

Property

- **Count**
DAO Methods by Object

This reference groups all DAO methods by object. To see whether a particular method is available for Microsoft Jet or ODBC workspaces, check the Help topic for that method.

**Connection**

- Container — no methods

**Database**

- Parameter — no methods
- Property — no methods

**QueryDef**
DBEngine

Recordset

Document

Relation

Error — no methods

TableDef

Field

User

Group
DAO Methods for Microsoft Jet Workspaces

This reference lists alphabetically all DAO methods available for Microsoft Jet workspaces (ISAM database files).
<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddNew</td>
<td>CreateDatabase</td>
</tr>
<tr>
<td>Append</td>
<td>CreateField</td>
</tr>
<tr>
<td>AppendChunk</td>
<td>CreateGroup</td>
</tr>
<tr>
<td>BeginTrans</td>
<td>CreateIndex</td>
</tr>
<tr>
<td>CancelUpdate</td>
<td>CreateProperty</td>
</tr>
<tr>
<td>Clone</td>
<td>CreateQueryDef</td>
</tr>
<tr>
<td>Close</td>
<td>CreateRelation</td>
</tr>
<tr>
<td>CommitTrans</td>
<td>CreateTableDef</td>
</tr>
<tr>
<td>CompactDatabase</td>
<td>CreateUser</td>
</tr>
<tr>
<td>CopyQueryDef</td>
<td>CreateWorkspace</td>
</tr>
<tr>
<td>Action</td>
<td>Action</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Delete</td>
<td>GetRows</td>
</tr>
<tr>
<td>Edit</td>
<td>Idle</td>
</tr>
<tr>
<td>Execute</td>
<td>MakeReplica</td>
</tr>
<tr>
<td>FillCache</td>
<td>Move</td>
</tr>
<tr>
<td>FindFirst</td>
<td>MoveFirst</td>
</tr>
<tr>
<td>FindLast</td>
<td>MoveLast</td>
</tr>
<tr>
<td>FindNext</td>
<td>MoveNext</td>
</tr>
<tr>
<td>FindPrevious</td>
<td>MovePrevious</td>
</tr>
<tr>
<td>GetChunk</td>
<td></td>
</tr>
</tbody>
</table>
N-Z

NewPassword    RepairDatabase
OpenDatabase    Requery
OpenRecordset   Rollback
PopulatePartial Seek
Refresh         SetOption
RefreshLink     Synchronize
RegisterDatabase Update
DAO Methods for ODBCDirect Workspaces

This reference alphabetically lists all DAO methods available for ODBCDirect workspaces.
A-C

AddNew
Append
AppendChunk
BeginTrans
Cancel
CancelUpdate

Clone
Close
CommitTrans
CreateQueryDef
CreateWorkspace
<table>
<thead>
<tr>
<th>D-M</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Delete</strong></td>
</tr>
<tr>
<td><strong>Edit</strong></td>
</tr>
<tr>
<td><strong>Execute</strong></td>
</tr>
<tr>
<td><strong>GetChunk</strong></td>
</tr>
<tr>
<td><strong>GetRows</strong></td>
</tr>
</tbody>
</table>
N-Z

NextRecordset  RegisterDatabase
OpenConnection  Requery
OpenDatabase  Rollback
OpenRecordset  Update
Refresh
AddNew Method

Creates a new record for an updatable Recordset object.

Syntax

\texttt{recordset.AddNew}

The recordset placeholder is an object variable that represents an updatable Recordset object to which you want to add a new record.

Remarks

Use the AddNew method to create and add a new record in the Recordset object named by recordset. This method sets the fields to default values, and if no default values are specified, it sets the fields to Null (the default values specified for a table-type Recordset).

After you modify the new record, use the Update method to save the changes and add the record to the Recordset. No changes occur in the database until you use the Update method.

Caution   If you issue an AddNew and then perform any operation that moves
to another record, but without using **Update**, your changes are lost without warning. In addition, if you close the **Recordset** or end the procedure that declares the **Recordset** or its **Database** object, the new record is discarded without warning.

**Note** When you use **AddNew** in a **Microsoft Jet workspace** and the database engine has to create a new **page** to hold the **current record**, page locking is **pessimistic**. If the new record fits in an existing page, page locking is **optimistic**.

If you haven't moved to the last record of your **Recordset**, records added to base tables by other processes may be included if they are positioned beyond the current record. If you add a record to your own **Recordset**, however, the record is visible in the **Recordset** and included in the underlying table where it becomes visible to any new **Recordset** objects.

The position of the new record depends on the type of **Recordset**:

- In a dynaset-type **Recordset** object, records are inserted at the end of the **Recordset**, regardless of any sorting or ordering rules that were in effect when the **Recordset** was opened.

- In a table-type **Recordset** object whose **Index** property has been set, records are returned in their proper place in the **sort order**. If you haven't set the **Index** property, new records are returned at the end of the **Recordset**.

The record that was current before you used **AddNew** remains current. If you want to make the new record current, you can set the **Bookmark** property to the **bookmark** identified by the **LastModified** property setting.

**Note** To add, edit, or delete a record, there must be a unique index on the record in the underlying data source. If not, a "Permission denied" error will occur on the **AddNew**, **Delete**, or **Edit** method call in a **Microsoft Jet workspace**, or an "Invalid argument" error will occur on the **Update** call in an **ODBCDirect workspace**.
Append Method

Adds a new DAO object to a collection.

Syntax

collection.Append object

The Append method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection</td>
<td>An object variable that represents any collection that can accept new objects (for limitations, see the table at the end of this topic).</td>
</tr>
<tr>
<td>object</td>
<td>An object variable that represents the object being appended, which must be of the same type as the elements of collection.</td>
</tr>
</tbody>
</table>

Remarks

You can use the Append method to add a new table to a database, add a field to a table, and add a field to an index.
The appended object becomes a **persistent object**, stored on disk, until you delete it by using the **Delete** method. If **collection** is a **Workspaces** collection (which is stored only in memory), the object is active until you remove it by using the **Close** method.

The addition of a new object occurs immediately, but you should use the **Refresh** method on any other collections that may be affected by changes to the database structure.

If the object you're appending isn't complete (such as when you haven't appended any **Field** objects to a **Fields** collection of an **Index** object before it’s appended to an **Indexes** collection) or if the properties set in one or more subordinate objects are incorrect, using the **Append** method causes an error. For example, if you haven’t specified a field type and then try to append the **Field** object to the **Fields** collection in a **TableDef** object, using the **Append** method triggers a run-time error.

The following table lists some limitations of the **Append** method. The object in the first column is an object containing the collection in the second column. The third column indicates whether you can append an object to that collection (for example, you can never append a **Container** object to the **Containers** collection of a **Database** object).

<table>
<thead>
<tr>
<th>Object</th>
<th>Collection</th>
<th>Can you append new objects?</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBEngine</td>
<td>Workspaces</td>
<td>Yes</td>
</tr>
<tr>
<td>DBEngine</td>
<td>Errors</td>
<td>No. New Error objects are automatically appended when they occur.</td>
</tr>
<tr>
<td>Workspace</td>
<td>Connections</td>
<td>No. Using the <strong>OpenConnection</strong> method automatically appends new objects.</td>
</tr>
<tr>
<td>Workspace</td>
<td>Databases</td>
<td>No. Using the <strong>OpenDatabase</strong> method automatically appends new objects.</td>
</tr>
<tr>
<td>Workspace</td>
<td>Groups</td>
<td>Yes</td>
</tr>
<tr>
<td>Workspace</td>
<td>Users</td>
<td>Yes</td>
</tr>
<tr>
<td>Connection</td>
<td>QueryDefs</td>
<td>No. Using the <strong>CreateQueryDef</strong> method automatically appends new objects.</td>
</tr>
<tr>
<td>Connection</td>
<td>Recordsets</td>
<td>No. Using the <a href="#">OpenRecordset</a> method automatically appends new objects.</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>Database</td>
<td>Containers</td>
<td>No</td>
</tr>
<tr>
<td>Database</td>
<td>QueryDefs</td>
<td>Only when the <a href="#">QueryDef</a> object is a new, unappended object created with no name. See the <a href="#">CreateQueryDef</a> method for details.</td>
</tr>
<tr>
<td>Database</td>
<td>Recordsets</td>
<td>No. Using the <a href="#">OpenRecordset</a> method automatically appends new objects.</td>
</tr>
<tr>
<td>Database</td>
<td>Relations</td>
<td>Yes</td>
</tr>
<tr>
<td>Database</td>
<td>TableDefs</td>
<td>Yes</td>
</tr>
<tr>
<td>Group</td>
<td>Users</td>
<td>Yes</td>
</tr>
<tr>
<td>User</td>
<td>Groups</td>
<td>Yes</td>
</tr>
<tr>
<td>Container</td>
<td>Documents</td>
<td>No</td>
</tr>
<tr>
<td>QueryDef</td>
<td>Fields</td>
<td>No</td>
</tr>
<tr>
<td>QueryDef</td>
<td>Parameters</td>
<td>No</td>
</tr>
<tr>
<td>Recordset</td>
<td>Fields</td>
<td>No</td>
</tr>
<tr>
<td>Relation</td>
<td>Fields</td>
<td>Yes</td>
</tr>
<tr>
<td>TableDef</td>
<td>Fields</td>
<td>Only when the <a href="#">Updatable</a> property of the <a href="#">TableDef</a> object is set to True, or when the <a href="#">TableDef</a> object is unappended.</td>
</tr>
<tr>
<td>TableDef</td>
<td>Indexes</td>
<td>Only when the <a href="#">Updatable</a> property of the <a href="#">TableDef</a> is set to True, or when the <a href="#">TableDef</a> object is unappended.</td>
</tr>
<tr>
<td>Index</td>
<td>Fields</td>
<td>Only when the <a href="#">Index</a> object is a new, unappended object.</td>
</tr>
<tr>
<td>Database, Field, Index, QueryDef, TableDef</td>
<td>Properties</td>
<td>Only when the <a href="#">Database</a>, <a href="#">Field</a>, <a href="#">Index</a>, <a href="#">QueryDef</a>, or <a href="#">TableDef</a> object is in a <a href="#">Microsoft Jet</a> workspace.</td>
</tr>
<tr>
<td>DBEngine, Parameter, Recordset, Workspace</td>
<td>Properties</td>
<td>No</td>
</tr>
</tbody>
</table>
AppendChunk Method

Appends data from a string expression to a Memo or Long Binary Field object in a Recordset.

Syntax

\[ \text{recordset} ! \text{field}.\text{AppendChunk} \ \text{source} \]

The AppendChunk method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>An object variable that represents the Recordset object containing the Fields collection.</td>
</tr>
<tr>
<td>field</td>
<td>An object variable that represents the name of a Field object whose Type property is set to dbMemo (Memo), dbLongBinary (Long Binary), or the equivalent.</td>
</tr>
<tr>
<td>source</td>
<td>A Variant (String subtype) expression or variable containing the data you want to append to the Field object specified by field.</td>
</tr>
</tbody>
</table>

Remarks
You can use the AppendChunk and GetChunk methods to access subsets of data in a Memo or Long Binary field.

You can also use these methods to conserve string space when you work with Memo and Long Binary fields. Certain operations (copying, for example) involve temporary strings. If string space is limited, you may need to work with chunks of a field instead of the entire field.

If there is no current record when you use AppendChunk, an error occurs.

Notes

- The initial AppendChunk operation (after an Edit or AddNew call) will simply place the data in the field, overwriting any existing data. Subsequent AppendChunk calls within the same Edit or AddNew session will then add to the existing data.

- In an ODBCDirect workspace, unless you first edit another field in the current record, using AppendChunk will fail (though no error occurs) while you are in Edit mode.

- In an ODBCDirect workspace, after you use AppendChunk on a field, you cannot read or write that field in an assignment statement until you move off the current record and then return to it. You can do this by using the MoveNext and MovePrevious methods.
BeginTrans, CommitTrans, Rollback Methods

The transaction methods manage transaction processing during a session defined by a Workspace object as follows:

- **BeginTrans** begins a new transaction.
- **CommitTrans** ends the current transaction and saves the changes.
- **Rollback** ends the current transaction and restores the databases in the Workspace object to the state they were in when the current transaction began.

**Syntax**

workspace.BeginTrans | CommitTrans [dbForceOSFlush] | Rollback

The workspace placeholder is an object variable that represents the Workspace containing the databases that will use transactions.
Remarks

You use these methods with a **Workspace** object when you want to treat a series of changes made to the databases in a session as one unit.

Typically, you use transactions to maintain the integrity of your data when you must both update records in two or more tables and ensure changes are completed (committed) in all tables or none at all (rolled back). For example, if you transfer money from one account to another, you might subtract an amount from one and add the amount to another. If either update fails, the accounts no longer balance. Use the **BeginTrans** method before updating the first record, and then, if any subsequent update fails, you can use the **Rollback** method to undo all of the updates. Use the **CommitTrans** method after you successfully update the last record.

In a Microsoft Jet workspace, you can include the **dbForceOSFlush** constant with **CommitTrans**. This forces the database engine to immediately flush all updates to disk, instead of caching them temporarily. Without using this option, a user could get control back immediately after the application program calls **CommitTrans**, turn the computer off, and not have the data written to disk. While using this option may affect your application’s performance, it is useful in situations where the computer could be shut off before cached updates are saved to disk.

**Caution**  Within one **Workspace** object, transactions are always global to the **Workspace** and aren't limited to only one **Connection** or **Database** object. If you perform operations on more than one connection or database within a **Workspace** transaction, resolving the transaction (that is, using the **CommitTrans** or **Rollback** method) affects all operations on all connections and databases within that workspace.

After you use **CommitTrans**, you can't undo changes made during that transaction unless the transaction is nested within another transaction that is itself rolled back. If you nest transactions, you must resolve the current transaction before you can resolve a transaction at a higher level of nesting.

If you want to have simultaneous transactions with overlapping, non-nested scopes, you can create additional **Workspace** objects to contain the concurrent transactions.
If you close a **Workspace** object without resolving any pending transactions, the transactions are automatically rolled back.

If you use the **CommitTrans** or **Rollback** method without first using the **BeginTrans** method, an error occurs.

Some ISAM databases used in a **Microsoft Jet workspace** may not support transactions, in which case the **Transactions** property of the **Database** object or **Recordset** object is **False**. To make sure the database supports transactions, check the value of the **Transactions** property of the **Database** object before using the **BeginTrans** method. If you are using a **Recordset** object based on more than one database, check the **Transactions** property of the **Recordset** object. If a **Recordset** is based entirely on **Microsoft Jet** tables, you can always use transactions. **Recordset** objects based on tables created by other database products, however, may not support transactions. For example, you can't use transactions in a **Recordset** based on a Paradox table. In this case, the **Transactions** property is **False**. If the **Database** or **Recordset** doesn't support transactions, the methods are ignored and no error occurs.

You can't nest transactions if you are accessing **ODBC data sources** through the Microsoft Jet database engine.

In ODBC workspaces, when you use **CommitTrans** your cursor may no longer be valid. Use the **Requery** method to view the changes in the **Recordset**, or close and re-open the **Recordset**.

**Notes**

- You can often improve the performance of your application by breaking operations that require disk access into transaction blocks. This buffers your operations and may significantly reduce the number of times a disk is accessed.

- In a Microsoft Jet workspace, transactions are logged in a file kept in the directory specified by the TEMP environment variable on the workstation. If the transaction log file exhausts the available storage on your TEMP drive, the database engine triggers a run-time error. At this point, if you use **CommitTrans**, an indeterminate number of operations are committed, but the remaining uncompleted operations are lost, and the operation has to be
restarted. Using a **Rollback** method releases the transaction log and rolls back all operations in the transaction.

- Closing a clone **Recordset** within a pending transaction will cause an implicit **Rollback** operation.
Cancel Method

Cancels execution of a pending asynchronous method call (ODBCDirect workspaces only).

Syntax

`object.Cancel`

The **Cancel** method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>A string expression that evaluates to one of the objects in the &quot;Applies To&quot; list.</td>
</tr>
</tbody>
</table>

Remarks

Use the **Cancel** method to terminate execution of an asynchronous **Execute**, **MoveLast**, **OpenConnection**, or **OpenRecordset** method call (that is, the method was invoked with the **dbRunAsync** option). **Cancel** will return a run-time error if **dbRunAsync** was not used in the method you're trying to terminate.

The following table shows what task is terminated when you use the **Cancel**
method on a particular type of object.

If *object* is a

| Connection | Execute or OpenConnection |
| QueryDef   | Execute                   |
| Recordset  | MoveLast or OpenRecordset |

This asynchronous method is terminated

An error will occur if, following a **Cancel** method call, you try to reference the object that would have been created by an asynchronous **OpenConnection** or **OpenRecordset** call (that is, the **Connection** or **Recordset** object from which you called the **Cancel** method).
CancelUpdate Method

Cancels any pending updates for a Recordset object.

Syntax

`recordset.CancelUpdate type`

The CancelUpdate method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recordset</code></td>
<td>An object variable that represents the Recordset object for which you are canceling pending updates.</td>
</tr>
<tr>
<td><code>type</code></td>
<td>Optional. A constant indicating the type of update, as specified in Settings.</td>
</tr>
</tbody>
</table>

Settings

You can use the following values for the type argument only if batch updating is enabled.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
</table>
**dbUpdateRegular** Default. Cancels pending changes that aren’t cached.

**dbUpdateBatch** Cancels pending changes in the update cache.

### Remarks

You can use the **CancelUpdate** method to cancel any pending updates resulting from an **Edit** or **AddNew** operation. For example, if a user invokes the **Edit** or **AddNew** method and hasn't yet invoked the **Update** method, **CancelUpdate** cancels any changes made after **Edit** or **AddNew** was invoked.

Check the **EditMode** property of the **Recordset** to determine if there is a pending operation that can be canceled.

**Note** Using the **CancelUpdate** method has the same effect as moving to another record without using the **Update** method, except that the current record doesn't change, and various properties, such as **BOF** and **EOF**, aren't updated.
Clone Method

Creates a duplicate Recordset object that refers to the original Recordset object.

Syntax

Set duplicate = original.Clone

The Clone method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duplicate</td>
<td>An object variable identifying the duplicate Recordset object you're creating.</td>
</tr>
<tr>
<td>original</td>
<td>An object variable identifying the Recordset object you want to duplicate.</td>
</tr>
</tbody>
</table>

Remarks

Use the Clone method to create multiple, duplicate Recordset objects. Each Recordset can have its own current record. Using Clone by itself doesn't change the data in the objects or in their underlying structures. When you use the Clone method, you can share bookmarks between two or more Recordset objects.
because their bookmarks are interchangeable.

You can use the **Clone** method when you want to perform an operation on a **Recordset** that requires multiple current records. This is faster and more efficient than opening a second **Recordset**. When you create a **Recordset** with the **Clone** method, it initially lacks a current record. To make a record current before you use the **Recordset** clone, you must set the **Bookmark** property or use one of the **Move** methods, one of the **Find** methods, or the **Seek** method.

Using the **Close** method on either the original or duplicate object doesn't affect the other object. For example, using **Close** on the original **Recordset** doesn't close the clone.

**Notes**

- Closing a clone **Recordset** within a pending transaction will cause an implicit **Rollback** operation.

- When you clone a table-type **Recordset** object in a **Microsoft Jet workspace**, the **Index** property setting is not cloned on the new copy of the **Recordset**. You must copy the **Index** property setting manually.

- You can use the **Clone** method with **forward-only–type Recordset objects** only in an **ODBCDirect workspace**.
Close Method

Closes an open DAO object.

Syntax

`object.Close`

The `object` placeholder is an object variable that represents an open Connection, Database, Recordset, or Workspace object.

Remarks

Closing an open object removes it from the collection to which it's appended. Any attempt to close the default workspace is ignored.

If the Connection, Database, Recordset, or Workspace object named by `object` is already closed when you use `Close`, a run-time error occurs.

Caution  If you exit a procedure that declares Connection, Database, or Recordset objects, those objects are closed, all pending transactions are rolled back, and any pending edits to your data are lost.
If you try to close a **Connection** or **Database** object while it has any open **Recordset** objects, the **Recordset** objects will be closed and any pending updates or edits will be canceled. Similarly, if you try to close a **Workspace** object while it has any open **Connection** or **Database** objects, those **Connection** and **Database** objects will be closed, which will close their **Recordset** objects.

Using the **Close** method on either an original or cloned **Recordset** object doesn't affect the other **Recordset** object.

To remove objects from updatable collections other than the **Connections**, **Databases**, **Recordsets**, and **Workspaces** collections, use the **Delete** method on those collections. You can't add a new member to the **Containers**, **Documents**, and **Errors** collections.

An alternative to the **Close** method is to set the value of an object variable to **Nothing** (Set dbsTemp = Nothing).
CompactDatabase Method

Copies and compacts a closed database, and gives you the option of changing its version, collating order, and encryption. (Microsoft Jet workspaces only).

Syntax

**DBEngine.CompactDatabase olddb, newdb, locale, options, password**

The **CompactDatabase** method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>olddb</strong></td>
<td>A <em>String</em> that identifies an existing, closed database. It can be a full path and file name, such as &quot;C:\db1.mdb&quot;. If the file name has an extension, you must specify it. If your network supports it, you can also specify a network path, such as &quot;\server1\share1\dir1\db1.mdb&quot;.</td>
</tr>
<tr>
<td><strong>newdb</strong></td>
<td>A <em>String</em> that is the file name (and path) of the compacted database that you're creating. You can also specify a network path. You can't use the <em>newdb</em> argument to specify the same database file as <em>olddb</em>.</td>
</tr>
<tr>
<td></td>
<td>Optional. A <em>Variant</em> that is a <em>string expression</em> that specifies</td>
</tr>
</tbody>
</table>
locale

a collating order for creating newdb, as specified in Settings. If you omit this argument, the locale of newdb is the same as olddb.

You can also create a password for newdb by concatenating the password string (starting with ";pwd=") with a constant in the locale argument, like this:

dbLangSpanish & ";pwd=NewPassword"

If you want to use the same locale as olddb (the default value), but specify a new password, simply enter a password string for locale:

";pwd=NewPassword"

Optional. A constant or combination of constants that indicates one or more options, as specified in Settings. You can combine options by summing the corresponding constants.

password

Optional. A Variant that is a string expression containing a password, if the database is password protected. The string ";pwd=" must precede the actual password. If you include a password setting in locale, this setting is ignored.

Settings

You can use one of the following constants for the locale argument to specify the CollatingOrder property for string comparisons of text.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Collating order</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbLangGeneral</td>
<td>English, German, French, Portuguese, Italian, and Modern Spanish</td>
</tr>
<tr>
<td>dbLangArabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>dbLangChineseSimplified</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>dbLangChineseTraditional</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>dbLangCyrillic</td>
<td>Russian</td>
</tr>
<tr>
<td>dbLangCzech</td>
<td>Czech</td>
</tr>
<tr>
<td>dbLangDutch</td>
<td>Dutch</td>
</tr>
<tr>
<td>dbLangGreek</td>
<td>Greek</td>
</tr>
<tr>
<td>dbLangHebrew</td>
<td>Hebrew</td>
</tr>
</tbody>
</table>
You can use one of the following constants in the *options* argument to specify whether to encrypt or to decrypt the database while it's compacted.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbEncrypt</td>
<td>Encrypt the database while compacting.</td>
</tr>
<tr>
<td>dbDecrypt</td>
<td>Decrypt the database while compacting.</td>
</tr>
</tbody>
</table>

If you omit an encryption constant or if you include both *dbDecrypt* and *dbEncrypt*, *newdb* will have the same encryption as *olddb*.

You can use one of the following constants in the *options* argument to specify the version of the data format for the compacted database. This constant affects only the version of the data format of *newdb* and doesn't affect the version of any Microsoft Access-defined objects, such as forms and reports.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbVersion10</td>
<td>Creates a database that uses the Microsoft Jet database engine version 1.0 file format while compacting.</td>
</tr>
<tr>
<td></td>
<td>Creates a database that uses the Microsoft Jet database engine version 1.0 file format while compacting.</td>
</tr>
</tbody>
</table>
dbVersion11  engine version 1.1 file format while compacting.

dbVersion20  Creates a database that uses the Microsoft Jet database engine version 2.0 file format while compacting.
              Creates a database that uses the Microsoft Jet database engine version 3.0 file format (compatible with version 3.5) while compacting.

You can specify only one version constant. If you omit a version constant, newdb will have the same version as olddb. You can compact newdb only to a version that is the same or later than that of olddb.

Remarks

As you change data in a database, the database file can become fragmented and use more disk space than is necessary. Periodically, you can use the CompactDatabase method to compact your database to defragment the database file. The compacted database is usually smaller and often runs faster.
You can also change the collating order, the encryption, or the version of the data format while you copy and compact the database.

You must close olddb before you compact it. In a multiuser environment, other users can't have olddb open while you're compacting it. If olddb isn't closed or isn't available for exclusive use, an error occurs.

Because CompactDatabase creates a copy of the database, you must have enough disk space for both the original and the duplicate databases. The compact operation fails if there isn't enough disk space available. The newdb duplicate database doesn't have to be on the same disk as olddb. After successfully compacting a database, you can delete the olddb file and rename the compacted newdb file to the original file name.

The CompactDatabase method copies all the data and the security permission settings from the database specified by olddb to the database specified by newdb.

If you use CompactDatabase to convert a version 1.x database to version 2.5 or 3.x, only applications using version Microsoft Jet 2.5 or 3.x can open the converted database.
Note  In an ODBCDirect workspace, using the CompactDatabase method doesn't return an error, but instead loads the Microsoft Jet database engine into memory.

Caution  Because the CompactDatabase method doesn't convert Microsoft Access objects, you shouldn't use CompactDatabase to convert a database containing such objects. To convert a database containing Microsoft Access objects, on the Tools menu, point to Database Utilities, and then click Convert Database.
CopyQueryDef Method

Returns a QueryDef object that is a copy of the QueryDef used to create the Recordset object represented by the recordset placeholder (Microsoft Jet workspaces only).

Syntax

Set querydef = recordset.CopyQueryDef

The CopyQueryDef method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>querydef</td>
<td>An object variable that represents the copy of a QueryDef object you want to create.</td>
</tr>
<tr>
<td>recordset</td>
<td>An object variable that represents the Recordset object created with the original QueryDef object.</td>
</tr>
</tbody>
</table>

Remarks

You can use the CopyQueryDef method to create a new QueryDef that is a duplicate of the QueryDef used to create the Recordset.
If a `QueryDef` wasn't used to create this `Recordset`, an error occurs. You must first open a `Recordset` with the `OpenRecordset` method before using the `CopyQueryDef` method.

This method is useful when you create a `Recordset` object from a `QueryDef`, and pass the `Recordset` to a function, and the function must re-create the SQL equivalent of the query, for example, to modify it in some way.
CreateDatabase Method

Creates a new Database object, saves the database to disk, and returns an opened Database object (Microsoft Jet workspaces only).

Syntax

Set database = workspace.CreateDatabase (name, locale, options)

The CreateDatabase method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>An object variable that represents the Database object you want to create.</td>
</tr>
<tr>
<td>workspace</td>
<td>An object variable that represents the existing Workspace object that will contain the database. If you omit workspace, CreateDatabase uses the default Workspace.</td>
</tr>
<tr>
<td>name</td>
<td>A String up to 255 characters long that is the name of the database file that you're creating. It can be the full path and file name, such as &quot;c:\db1.mdb&quot;. If you don't supply a file name extension, .mdb is appended. If your network supports it, you can also specify a network path, such as</td>
</tr>
</tbody>
</table>
"\\server1\share1\dir1\db1". You can only create .mdb database files with this method.

**locale**

A string expression that specifies a collating order for creating the database, as specified in Settings. You must supply this argument or an error occurs.

You can also create a password for the new Database object by concatenating the password string (starting with ";pwd=") with a constant in the locale argument, like this:

dbLangSpanish & ";pwd=NewPassword"

If you want to use the default locale, but specify a password, simply enter a password string for the locale argument:

";pwd=NewPassword"

**options**

Optional. A constant or combination of constants that indicates one or more options, as specified in Settings. You can combine options by summing the corresponding constants.

**Settings**

You can use one of the following constants for the locale argument to specify the CollatingOrder property of text for string comparisons.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Collating order</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbLangGeneral</td>
<td>English, German, French, Portuguese, Italian, and Modern Spanish</td>
</tr>
<tr>
<td>dbLangArabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>dbLangChineseSimplified</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>dbLangChineseTraditional</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>dbLangCyrillic</td>
<td>Russian</td>
</tr>
<tr>
<td>dbLangCzech</td>
<td>Czech</td>
</tr>
<tr>
<td>dbLangDutch</td>
<td>Dutch</td>
</tr>
<tr>
<td>dbLangGreek</td>
<td>Greek</td>
</tr>
<tr>
<td>dbLangHebrew</td>
<td>Hebrew</td>
</tr>
<tr>
<td>dbLangHungarian</td>
<td>Hungarian</td>
</tr>
<tr>
<td>dbLangIcelandic</td>
<td>Icelandic</td>
</tr>
</tbody>
</table>
You can use one or more of the following constants in the *options* argument to specify which version the data format should have and whether or not to encrypt the database.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbEncrypt</code></td>
<td>Creates an encrypted database.</td>
</tr>
<tr>
<td><code>dbVersion10</code></td>
<td>Creates a database that uses the <a href="https://msdn.microsoft.com/en-us/library/aa365202(v=vs.60).aspx">Microsoft Jet database engine</a> version 1.0 file format.</td>
</tr>
<tr>
<td><code>dbVersion11</code></td>
<td>Creates a database that uses the Microsoft Jet database engine version 1.1 file format.</td>
</tr>
<tr>
<td><code>dbVersion20</code></td>
<td>Creates a database that uses the Microsoft Jet database engine version 2.0 file format. (Default)</td>
</tr>
<tr>
<td><code>dbVersion30</code></td>
<td>(Default) Creates a database that uses the Microsoft Jet database engine version 3.0 file format (compatible with version 3.5).</td>
</tr>
</tbody>
</table>

If you omit the encryption constant, **CreateDatabase** creates an un-encrypted database. You can specify only one version constant. If you omit a version constant, **CreateDatabase** creates a database that uses the Microsoft Jet database engine version 3.0 file format.

**Remarks**
Use the **CreateDatabase** method to create and open a new, empty database, and return the **Database** object. You must complete its structure and content by using additional **DAO objects**. If you want to make a partial or complete copy of an existing database, you can use the **CompactDatabase** method to make a copy that you can customize.
CreateField Method

Creates a new Field object (Microsoft Jet workspaces only).

Syntax

Set field = object.CreateField (name, type, size)

The CreateField method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>An object variable that represents the Field object you want to create. An object variable that represents the Index, Relation, or TableDef object for which you want to create the new Field object.</td>
</tr>
<tr>
<td>object</td>
<td>Optional. A Variant (String subtype) that uniquely names the new Field object. See the Name property for details on valid Field names.</td>
</tr>
<tr>
<td>name</td>
<td>Optional. A constant that determines the data type of the new Field object. See the Type property for valid data types.</td>
</tr>
</tbody>
</table>
Optional. A **Variant** (**Integer** subtype) that indicates the maximum size, in bytes, of a **Field** object that contains text. See the **Size** property for valid size values. This argument is ignored for numeric and fixed-width fields.

**Remarks**

You can use the **CreateField** method to create a new field, as well as specify the name, data type, and size of the field. If you omit one or more of the optional parts when you use **CreateField**, you can use an appropriate assignment statement to set or reset the corresponding property before you append the new object to a collection. After you append the new object, you can alter some but not all of its property settings. See the individual property topics for more details.

The **type** and **size** arguments apply only to **Field** objects in a **TableDef** object. These arguments are ignored when a **Field** object is associated with an **Index** or **Relation** object.

If **name** refers to an object that is already a member of the collection, a run-time error occurs when you use the **Append** method.

To remove a **Field** object from a **Fields** collection, use the **Delete** method on the collection. You can't delete a **Field** object from a **TableDef** object's **Fields** collection after you create an **index** that references the field.
CreateGroup Method

Creates a new Group object (Microsoft Jet workspaces only).

Syntax

Set group = object.CreateGroup (name, pid)

The CreateGroup method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>An object variable that represents the Group you want to create.</td>
</tr>
<tr>
<td>object</td>
<td>An object variable that represents the User or Workspace object for which you want to create the new Group object. Optional. A Variant (String subtype) that uniquely names the new Group object. See the Name property for details on valid Group names.</td>
</tr>
<tr>
<td>name</td>
<td>Optional. A Variant (String subtype) containing the PID of a group account. The identifier must contain from 4 to 20 alphanumeric characters. See the PID property for more information on valid personal identifiers.</td>
</tr>
<tr>
<td>pid</td>
<td></td>
</tr>
</tbody>
</table>
Remarks

You can use the CreateGroup method to create a new Group object for a User or Workspace. If you omit one or both of the optional parts when you use CreateGroup, you can use an appropriate assignment statement to set or reset the corresponding property before you append the new object to a collection. After you append the object, you can alter some but not all of its property settings. See the individual property topics for more details.

If name refers to an object that is already a member of the collection, a run-time error occurs when you use the Append method.

To remove a Group object from a collection, use the Delete method on the Groups collection.
CreateIndex Method

Creates a new Index object (Microsoft Jet workspaces only).

Syntax

Set index = tabledef.CreateIndex (name)

The CreateIndex method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>An object variable that represents the index you want to create.</td>
</tr>
<tr>
<td>tabledef</td>
<td>An object variable that represents the TableDef object you want to use to create the new Index object.</td>
</tr>
<tr>
<td>name</td>
<td>Optional. A Variant (String subtype) that uniquely names the new Index object. See the Name property for details on valid Index names.</td>
</tr>
</tbody>
</table>

Remarks

You can use the CreateIndex method to create a new Index object for a
TableDef object. If you omit the optional name part when you use CreateIndex, you can use an appropriate assignment statement to set or reset the Name property before you append the new object to a collection. After you append the object, you may or may not be able to set its Name property, depending on the type of object that contains the Indexes collection. See the Name property topic for more details.

If name refers to an object that is already a member of the collection, a run-time error occurs when you use the Append method.

To remove an Index object from a collection, use the Delete method on the collection.
CreateProperty Method

Creates a new user-defined Property object (Microsoft Jet workspaces only).

Syntax

Set property = object.CreateProperty (name, type, value, DDL)

The CreateProperty method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property</td>
<td>An object variable that represents the Property object you want to create.</td>
</tr>
<tr>
<td>object</td>
<td>An object variable that represents the Database, Field, Index, QueryDef, Document, or TableDef object you want to use to create the new Property object.</td>
</tr>
<tr>
<td>name</td>
<td>Optional. A Variant (String subtype) that uniquely names the new Property object. See the Name property for details on valid Property names.</td>
</tr>
<tr>
<td>type</td>
<td>Optional. A constant that defines the data type of the new Property object. See the Type property for valid data types.</td>
</tr>
</tbody>
</table>
value  Optional. A **Variant** containing the initial property value. See the **Value** property for details.

Optional. A **Variant** (**Boolean** subtype) that indicates whether or not the **Property** is a **DDL** object. The default is **False**. If **DDL** is **True**, users can't change or delete this **Property** object unless they have **dbSecWriteDef** permission.

**Remarks**

You can create a user-defined **Property** object only in the **Properties** collection of an object that is **persistent**.

If you omit one or more of the optional parts when you use **CreateProperty**, you can use an appropriate assignment statement to set or reset the corresponding property before you append the new object to a collection. After you append the object, you can alter some but not all of its property settings. See the **Name**, **Type**, and **Value** property topics for more details.

If **name** refers to an object that is already a member of the collection, a run-time error occurs when you use the **Append** method.

To remove a user-defined **Property** object from the collection, use the **Delete** method on the **Properties** collection. You can't delete built-in properties.

**Note**  If you omit the **DDL** argument, it defaults to **False** (non-**DDL**). Because no corresponding DDL property is exposed, you must delete and re-create a **Property** object you want to change from DDL to non-DDL.
CreateQueryDef Method

Creates a new QueryDef object in a specified Connection or Database object.

Syntax

Set querydef = object.CreateQueryDef (name, sqltext)

The CreateQueryDef method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>querydef</td>
<td>An object variable that represents the QueryDef object you want to create.</td>
</tr>
<tr>
<td>object</td>
<td>An object variable that represents an open Connection or Database object that will contain the new QueryDef.</td>
</tr>
<tr>
<td>name</td>
<td>Optional. A Variant (String subtype) that uniquely names the new QueryDef.</td>
</tr>
<tr>
<td>sqltext</td>
<td>Optional. A Variant (String subtype) that is an SQL statement defining the QueryDef. If you omit this argument, you can define the QueryDef by setting its SQL property before or after you append it to a collection.</td>
</tr>
</tbody>
</table>
**Remarks**

In a Microsoft Jet workspace, if you provide anything other than a zero-length string for the name when you create a `QueryDef`, the resulting `QueryDef` object is automatically appended to the `QueryDefs` collection. In an ODBCDirect workspace, `QueryDef` objects are always temporary.

In an ODBCDirect workspace, the `sqltext` argument can specify an SQL statement or a Microsoft SQL Server stored procedure and its parameters.

If the object specified by `name` is already a member of the `QueryDefs` collection, a run-time error occurs. You can create a temporary `QueryDef` by using a zero-length string for the `name` argument when you execute the `CreateQueryDef` method. You can also accomplish this by setting the `Name` property of a newly created `QueryDef` to a zero-length string ("""). Temporary `QueryDef` objects are useful if you want to repeatedly use dynamic SQL statements without having to create any new permanent objects in the `QueryDefs` collection. You can't append a temporary `QueryDef` to any collection because a zero-length string isn't a valid name for a permanent `QueryDef` object. You can always set the `Name` and `SQL` properties of the newly created `QueryDef` object and subsequently append the `QueryDef` to the `QueryDefs` collection.

To run the SQL statement in a `QueryDef` object, use the `Execute` or `OpenRecordset` method.

Using a `QueryDef` object is the preferred way to perform SQL pass-through queries with ODBC databases.

To remove a `QueryDef` object from a `QueryDefs` collection in a Microsoft Jet database, use the `Delete` method on the collection. For an ODBCDirect database, use the `Close` method on the `QueryDef` object.
CreateRelation Method

Creates a new Relation object (Microsoft Jet workspaces only).

Syntax

Set relation = database.CreateRelation (name, table, foreigntable, attributes)

The CreateRelation method syntax uses these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relation</td>
<td>An object variable that represents the Relation object you want to create.</td>
</tr>
<tr>
<td>database</td>
<td>An object variable that represents the Database object for which you want to create the new Relation object.</td>
</tr>
<tr>
<td>name</td>
<td>Optional. A Variant (String subtype) that uniquely names the new Relation object. See the Name property for details on valid Relation names.</td>
</tr>
<tr>
<td>table</td>
<td>Optional. A Variant (String subtype) that names the primary table in the relation. If the table doesn't exist before you append the Relation object, a run-time error occurs.</td>
</tr>
</tbody>
</table>
foreigntable
Optional. A Variant (String subtype) that names the foreign table in the relation. If the table doesn't exist before you append the Relation object, a run-time error occurs.

attributes
Optional. A constant or combination of constants that contains information about the relationship type. See the Attributes property for details.

Remarks

The Relation object provides information to the Microsoft Jet database engine about the relationship between fields in two TableDef or QueryDef objects. You can implement referential integrity by using the Attributes property.

If you omit one or more of the optional parts when you use the CreateRelation method, you can use an appropriate assignment statement to set or reset the corresponding property before you append the new object to a collection. After you append the object, you can't alter any of its property settings. See the individual property topics for more details.

Before you can use the Append method on a Relation object, you must append the appropriate Field objects to define the primary and foreign key relationship tables.

If name refers to an object that is already a member of the collection or if the Field object names provided in the subordinate Fields collection are invalid, a run-time error occurs when you use the Append method.

You can't establish or maintain a relationship between a replicated table and a local table.

To remove a Relation object from the Relations collection, use the Delete method on the collection.
CreateTableDef Method

Creates a new TableDef object (Microsoft Jet workspaces only).

Syntax

Set tabledef = database.CreateTableDef (name, attributes, source, connect)

The CreateTableDef method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabledef</td>
<td>An object variable that represents the TableDef object you want to create.</td>
</tr>
<tr>
<td>database</td>
<td>An object variable that represents the Database object you want to use to create the new TableDef object.</td>
</tr>
<tr>
<td>name</td>
<td>Optional. A Variant (String subtype) that uniquely names the new TableDef object. See the Name property for details on valid TableDef names.</td>
</tr>
<tr>
<td>attributes</td>
<td>Optional. A constant or combination of constants that indicates one or more characteristics of the new TableDef object. See the Attributes property for more information.</td>
</tr>
</tbody>
</table>
source

Optional. A **Variant** (**String** subtype) containing the name of a table in an external database that is the original source of the data. The `source` string becomes the `SourceTableName` property setting of the new **TableDef** object.

Optional. A **Variant** (**String** subtype) containing information about the source of an open database, a database used in a **pass-through query**, or a **linked table**. See the `Connect` property for more information about valid connection strings.

**Remarks**

If you omit one or more of the optional parts when you use the **CreateTableDef** method, you can use an appropriate assignment statement to set or reset the corresponding property before you append the new object to a collection. After you append the object, you can alter some but not all of its properties. See the individual property topics for more details.

If `name` refers to an object that is already a member of the collection, or you specify an invalid property in the **TableDef** or **Field** object you're appending, a run-time error occurs when you use the **Append** method. Also, you can't append a **TableDef** object to the **TableDefs** collection until you define at least one **Field** for the **TableDef** object.

To remove a **TableDef** object from the **TableDefs** collection, use the **Delete** method on the collection.
CreateUser Method

Creates a new User object (Microsoft Jet workspaces only).

Syntax

Set user = object.CreateUser (name, pid, password)

The CreateUser method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>An object variable that represents the User object you want to create.</td>
</tr>
<tr>
<td>object</td>
<td>An object variable that represents the Group or Workspace object for which you want to create the new User object. Optional. A Variant (String subtype) that uniquely names the new User object. See the Name property for details on valid User names.</td>
</tr>
<tr>
<td>name</td>
<td>Optional. A Variant (String subtype) containing the PID of a user account. The identifier must contain from 4 to 20 alphanumeric characters. See the PID property for more information on valid personal identifiers.</td>
</tr>
</tbody>
</table>
Optional. A **Variant (String subtype)** containing the password for the new **User** object. The password can be up to 14 characters long and can include any characters except the **ASCII** character 0 (null). See the **Password** property for more information on valid passwords.

**Remarks**

If you omit one or more of the optional parts when you use the **CreateUser** method, you can use an appropriate assignment statement to set or reset the corresponding property before you append the new object to a collection. After you append the object, you can alter some but not all of its property settings. See the **PID**, **Name**, and **Password** property topics for more details.

If `name` refers to an object that is already a member of the collection, a run-time error occurs when you use the **Append** method.

To remove a **User** object from the **Users** collection, use the **Delete** method on the collection.
CreateWorkspace Method

Creates a new **Workspace** object.

**Syntax**

```
Set workspace = CreateWorkspace(name, user, password, type)
```

The **CreateWorkspace** method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>workspace</strong></td>
<td>An <strong>object variable</strong> that represents the <strong>Workspace</strong> object you want to create.</td>
</tr>
<tr>
<td></td>
<td>A <strong>String</strong> that uniquely names the new <strong>Workspace</strong> object. See the <strong>Name</strong> property for details on valid <strong>Workspace</strong> names.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td></td>
</tr>
<tr>
<td><strong>user</strong></td>
<td></td>
</tr>
<tr>
<td><strong>password</strong></td>
<td>A <strong>String</strong> containing the password for the new <strong>Workspace</strong> object. The password can be up to 14 characters long and can include any characters except</td>
</tr>
</tbody>
</table>
Optional. A constant that indicates the type of workspace, as described in Settings.

**Settings**

You can use the following constants for *type*.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbUseJet</td>
<td>Creates a <a href="#">Microsoft Jet workspace</a>.</td>
</tr>
<tr>
<td>dbUseODBC</td>
<td>Creates an <a href="#">ODBCDirect workspace</a>.</td>
</tr>
</tbody>
</table>

**Remarks**

Once you use the `CreateWorkspace` method to create a new `Workspace` object, a `Workspace` *session* is started, and you can refer to the `Workspace` object in your application.

`Workspace` objects aren't permanent, and you can't save them to disk. Once you create a `Workspace` object, you can't alter any of its property settings, except for the `Name` property, which you can modify before appending the `Workspace` object to the `Workspaces` collection.

You don't have to append the new `Workspace` object to a collection before you can use it. You append a newly created `Workspace` object only if you need to refer to it through the `Workspaces` collection.

The *type* option determines whether the new `Workspace` is a Microsoft Jet or ODBCDirect workspace. If you set *type* to `dbUseODBC` and you haven't already created any Microsoft Jet workspaces, then the Microsoft Jet database engine will not be loaded into memory, and all activity will occur with the ODBC data source subsequently identified in a `Connection` object. If you omit *type*, the `DefaultType` property of `DBEngine` will determine which type of data source the `Workspace` is connected to. You can have both Microsoft Jet and ODBCDirect workspaces open at the same time.
To remove a **Workspace** object from the **Workspaces** collection, close all open databases and connections and then use the **Close** method on the **Workspace** object.
Delete Method

- **Recordset** objects deletes the **current record** in an updatable **Recordset** object. For **ODBCDirect workspaces**, the type of driver determines whether **Recordset** objects are updatable and therefore support the **Delete** method.

- Collections deletes a **persistent object** from a **collection**.

**Syntax**

```
recordset.Delete

collection.Delete objectname
```

The **Delete** method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recordset</code></td>
<td>An <strong>object variable</strong> that represents an updatable <strong>Recordset</strong> object containing the record you want to delete.</td>
</tr>
<tr>
<td><code>collection</code></td>
<td>An object variable that represents a collection from which you are deleting <strong>objectname</strong>.</td>
</tr>
<tr>
<td><code>objectname</code></td>
<td>A <strong>String</strong> that is the <strong>Name</strong> property setting of an object in</td>
</tr>
</tbody>
</table>
Remarks

You can use the **Delete** method to delete a current record from a **Recordset** or a member from a collection, such as a stored table from a database, a stored field from a table, or a stored index from a table.

Recordsets

A **Recordset** must contain a current record before you use **Delete**; otherwise, a run-time error occurs.

In an updatable **Recordset** object, **Delete** removes the current record and makes it inaccessible. Although you can't edit or use the deleted record, it remains current. Once you move to another record, however, you can't make the deleted record current again. Subsequent references to a deleted record in a **Recordset** are invalid and produce an error.

You can undo a record deletion if you use **transactions** and the **Rollback** method.

If the **base table** is the **primary table** in a **cascading delete** relationship, deleting the current record may also delete one or more records in a **foreign table**.

**Note**  To add, edit, or delete a record, there must be a unique index on the record in the underlying data source. If not, a "Permission denied" error will occur on the **AddNew**, **Delete**, or **Edit** method call in a **Microsoft Jet workspace**, or an "Invalid argument" error will occur on the **Update** method call in an **ODBCDirect workspace**.

Collections

You can use the **Delete** method to delete a persistent object. However, if the collection is a **Databases**, **Recordsets**, or **Workspaces** collection (each of which is stored only in memory), you can remove an open or active object only by closing that object with the **Close** method.

The deletion of a stored object occurs immediately, but you should use the **Refresh** method on any other collections that may be affected by changes to the
When you delete a TableDef object from the TableDefs collection, you delete the table definition and the data in the table.

The following table lists some limitations of the Delete method. The object in the first column contains the collection in the second column. The third column indicates if you can delete an object from that collection (for example, you can never delete a Container object from the Containers collection of a Database object).

<table>
<thead>
<tr>
<th>Object</th>
<th>Collection</th>
<th>Can you use the Delete method?</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBEngine</td>
<td>Workspaces</td>
<td>No. Closing the objects deletes them.</td>
</tr>
<tr>
<td>DBEngine</td>
<td>Errors</td>
<td>No</td>
</tr>
<tr>
<td>Workspace</td>
<td>Connections</td>
<td>No. Closing the objects deletes them.</td>
</tr>
<tr>
<td>Workspace</td>
<td>Databases</td>
<td>No. Closing the objects deletes them.</td>
</tr>
<tr>
<td>Workspace</td>
<td>Groups</td>
<td>Yes</td>
</tr>
<tr>
<td>Workspace</td>
<td>Users</td>
<td>Yes</td>
</tr>
<tr>
<td>Connection</td>
<td>QueryDefs</td>
<td>No</td>
</tr>
<tr>
<td>Connection</td>
<td>Recordsets</td>
<td>No. Closing the objects deletes them.</td>
</tr>
<tr>
<td>Database</td>
<td>Containers</td>
<td>No</td>
</tr>
<tr>
<td>Database</td>
<td>QueryDefs</td>
<td>Yes</td>
</tr>
<tr>
<td>Database</td>
<td>Recordsets</td>
<td>No. Closing the objects deletes them.</td>
</tr>
<tr>
<td>Database</td>
<td>Relations</td>
<td>Yes</td>
</tr>
<tr>
<td>Database</td>
<td>TableDefs</td>
<td>Yes</td>
</tr>
<tr>
<td>Group</td>
<td>Users</td>
<td>Yes</td>
</tr>
<tr>
<td>User</td>
<td>Groups</td>
<td>Yes</td>
</tr>
<tr>
<td>Container</td>
<td>Documents</td>
<td>No</td>
</tr>
<tr>
<td>QueryDef</td>
<td>Fields</td>
<td>No</td>
</tr>
<tr>
<td>QueryDef</td>
<td>Parameters</td>
<td>No</td>
</tr>
<tr>
<td>Recordset</td>
<td>Fields</td>
<td>No</td>
</tr>
<tr>
<td>Object</td>
<td>Fields</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Relation</td>
<td>Fields</td>
<td>Only when the <strong>Relation</strong> object is a new, unappended object.</td>
</tr>
<tr>
<td>TableDef</td>
<td>Fields</td>
<td>Only when the <strong>TableDef</strong> object is new and hasn’t been appended to the database, or when the <strong>Updatable</strong> property of the <strong>TableDef</strong> is set to <strong>True</strong>.</td>
</tr>
<tr>
<td>TableDef</td>
<td>Indexes</td>
<td>Only when the <strong>TableDef</strong> object is new and hasn’t been appended to the database, or when the <strong>Updatable</strong> property of the <strong>TableDef</strong> is set to <strong>True</strong>.</td>
</tr>
<tr>
<td>Index</td>
<td>Fields</td>
<td>Only when the <strong>Index</strong> object is new and hasn’t been appended to the database.</td>
</tr>
<tr>
<td>Database, Field, Index, QueryDef, TableDef</td>
<td>Properties</td>
<td>Only when the property is user-defined.</td>
</tr>
<tr>
<td>DBEngine, Parameter, Recordset, Workspace</td>
<td>Properties</td>
<td>No</td>
</tr>
</tbody>
</table>
Edit Method

Copies the current record from an updatable Recordset object to the copy buffer for subsequent editing.

Syntax

`recordset.Edit`

The recordset placeholder represents an open, updatable Recordset object that contains the record you want to edit.

Remarks

Once you use the Edit method, changes made to the current record's fields are copied to the copy buffer. After you make the desired changes to the record, use the Update method to save your changes.

The current record remains current after you use Edit.

Caution  If you edit a record and then perform any operation that moves to another record, but without first using Update, your changes are lost without warning. In addition, if you close recordset or end the procedure which declares
the **Recordset** or the parent **Database** or **Connection** object, your edited record is discarded without warning.

Using **Edit** produces an error if:

- There is no current record.
- The **Connection**, **Database**, or **Recordset** object was opened as read-only.
- No fields in the record are updatable.
- The **Database** or **Recordset** was opened for exclusive use by another user (**Microsoft Jet workspace**).
- Another user has locked the page containing your record (**Microsoft Jet workspace**).

In a Microsoft Jet workspace, when the **Recordset** object's **LockEdits** property setting is **True** (**pessimistically** locked) in a multiuser environment, the record remains locked from the time **Edit** is used until the update is complete. If the **LockEdits** property setting is **False** (**optimistically** locked), the record is locked and compared with the pre-edited record just before it's updated in the database. If the record has changed since you used the **Edit** method, the **Update** operation fails with a run-time error if you use **OpenRecordset** without specifying **dbSeeChanges**. By default, Microsoft Jet-connected **ODBC** and **installable ISAM** databases always use optimistic locking.

In an **ODBCDirect workspace**, once you edit (and use **Update** to update) a record’s **primary key** field, you can no longer edit fields in that record until you close the **Recordset**, and then retrieve the record again in a subsequent query.

**Note**  To add, edit, or delete a record, there must be a unique index on the record in the underlying data source. If not, a "Permission denied" error will occur on the **AddNew**, **Delete**, or **Edit** method call in a Microsoft Jet workspace, or an "Invalid argument" error will occur on the **Update** call in an ODBCDirect workspace.
Execute Method

Runs an action query or executes an SQL statement on a specified Connection or Database object.

Syntax

object.Execute source, options

querydef.Execute options

The Execute method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>A Connection or Database object variable on which the query will run.</td>
</tr>
<tr>
<td></td>
<td>An object variable that represents the QueryDef object whose SQL property setting specifies the SQL statement to execute.</td>
</tr>
<tr>
<td>querydef</td>
<td>A String that is an SQL statement or the Name property value of a QueryDef object.</td>
</tr>
<tr>
<td>source</td>
<td>Optional. A constant or combination of constants that</td>
</tr>
</tbody>
</table>
options determines the data integrity characteristics of the query, as specified in Settings.

Settings

You can use the following constants for options.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbDenyWrite</td>
<td>Denies write permission to other users (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbInconsistent</td>
<td>(Default) Executes inconsistent updates (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbConsistent</td>
<td>Executes consistent updates (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbSQLPassThrough</td>
<td>Executes an SQL pass-through query. Setting this option passes the SQL statement to an ODBC database for processing (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbFailOnError</td>
<td>Rolls back updates if an error occurs (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbSeeChanges</td>
<td>Generates a run-time error if another user is changing data you are editing (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td>dbRunAsync</td>
<td>Executes the query asynchronously (ODBCDirect Connection and QueryDef objects only).</td>
</tr>
<tr>
<td>dbExecDirect</td>
<td>Executes the statement without first calling SQLPrepare ODBC API function (ODBCDirect Connection and QueryDef objects only).</td>
</tr>
</tbody>
</table>

Note  The constants dbConsistent and dbInconsistent are mutually exclusive. You can use one or the other, but not both in a given instance of OpenRecordset. Using both dbConsistent and dbInconsistent causes an error.

Remarks

The Execute method is valid only for action queries. If you use Execute with another type of query, an error occurs. Because an action query doesn't return
any records, **Execute** doesn't return a **Recordset**. (Executing an SQL pass-through query in an ODBCDirect workspace will not return an error if a **Recordset** isn't returned.)

Use the **RecordsAffected** property of the **Connection**, **Database**, or **QueryDef** object to determine the number of records affected by the most recent **Execute** method. For example, **RecordsAffected** contains the number of records deleted, updated, or inserted when executing an action query. When you use the **Execute** method to run a query, the **RecordsAffected** property of the **QueryDef** object is set to the number of records affected.

In a Microsoft Jet workspace, if you provide a syntactically correct SQL statement and have the appropriate permissions, the **Execute** method won't fail — even if not a single row can be modified or deleted. Therefore, always use the **dbFailOnError** option when using the **Execute** method to run an update or delete query. This option generates a run-time error and rolls back all successful changes if any of the records affected are locked and can't be updated or deleted.

In earlier versions of the Microsoft Jet Database Engine, SQL statements were automatically embedded in implicit transactions. If part of a statement executed with **dbFailOnError** failed, the entire statement would be rolled back. To improve performance, these implicit transactions were removed starting with version 3.5. If you are updating older DAO code, be sure to consider using explicit transactions around **Execute** statements.

For best performance in a Microsoft Jet workspace, especially in a multiuser environment, nest the **Execute** method inside a **transaction**. Use the **BeginTrans** method on the current **Workspace** object, then use the **Execute** method, and complete the transaction by using the **CommitTrans** method on the **Workspace**. This saves changes on disk and frees any locks placed while the query is running.

In an **ODBCDirect workspace**, if you include the optional **dbRunAsync** constant, the query runs asynchronously. To determine whether an asynchronous query is still executing, check the value of the **StillExecuting** property on the object from which the **Execute** method was called. To terminate execution of an asynchronous **Execute** method call, use the **Cancel** method.

In an ODBCDirect workspace, use single quotes to signify strings or an error
occurs.
## FillCache Method

Fills all or a part of a local cache for a `Recordset` object that contains data from a Microsoft Jet-connected ODBC data source (Microsoft Jet-connected ODBC databases only).

### Syntax

```vba
recordset.FillCache rows, startbookmark
```

The **FillCache** method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recordset</code></td>
<td>An object variable that represents a <code>Recordset</code> object created from an ODBC data source, such as a <code>TableDef</code> representing a linked table or a <code>QueryDef</code> object derived from such a <code>TableDef</code>. Optional. A <code>Variant</code> (Integer subtype) that specifies the number of rows to store in the cache. If you omit this argument, the value is determined by the <code>CacheSize</code> property setting.</td>
</tr>
<tr>
<td><code>rows</code></td>
<td>Optional. A <code>Variant</code> (String subtype) that specifies a</td>
</tr>
</tbody>
</table>
The cache is filled starting from the record indicated by this bookmark. If you omit this argument, the cache is filled starting from the record indicated by the CacheStart property.

**Remarks**

Caching improves the performance of an application that retrieves data from a remote server. A cache is space in local memory that holds the data most recently retrieved from the server; this assumes that the data will probably be requested again while the application is running. When a user requests data, the Microsoft Jet database engine checks the cache for the data first rather than retrieving it from the server, which takes more time. The cache doesn't save data that doesn't come from an ODBC data source.

Rather than waiting for the cache to be filled with records as they are retrieved, you can use the FillCache method to explicitly fill the cache at any time. This is a faster way to fill the cache because FillCache retrieves several records at once instead of one at a time. For example, while you view each screenful of records, your application uses FillCache to retrieve the next screenful of records for viewing.

Any Microsoft Jet-connected ODBC data source that you access with Recordset objects can have a local cache. To create the cache, open a Recordset object from the remote data source, and then set the CacheSize and CacheStart properties of the Recordset.

If rows and startbookmark create a range of records that is partially or entirely outside the range of records specified by the CacheSize and CacheStart properties, the portion of the recordset outside this range is ignored and will not be loaded into the cache.

If FillCache requests more records than the number remaining in the remote data source, Microsoft Jet retrieves only the remaining records, and no error occurs.

**Notes**

- Records retrieved from the cache don't reflect concurrent changes that other users made to the source data.
• **FillCache** only retrieves records not already cached. To force an update of all the cached data, set the **CacheSize** property of the **Recordset** to 0, reset it to the size of the cache you originally requested, and then use **FillCache**.
FindFirst, FindLast, FindNext, FindPrevious Methods

Locates the first, last, next, or previous record in a dynaset- or snapshot-type Recordset object that satisfies the specified criteria and makes that record the current record (Microsoft Jet workspaces only).

Syntax

recordset.{FindFirst | FindLast | FindNext | FindPrevious} criteria

The Find methods have these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>An object variable that represents an existing dynaset- or snapshot-type Recordset object. A String used to locate the record. It is like the</td>
</tr>
</tbody>
</table>
\textit{criteria}

\texttt{WHERE} clause in an \texttt{SQL statement}, but without the word \texttt{WHERE}.

\textbf{Remarks}

If you want to include all the records in your search — not just those that meet a specific condition — use the \texttt{Move} methods to move from record to record. To locate a record in a table-type \texttt{Recordset}, use the \texttt{Seek} method.

If a record matching the criteria isn't located, the current record pointer is unknown, and the \texttt{NoMatch} property is set to \texttt{True}. If \texttt{recordset} contains more than one record that satisfies the criteria, \texttt{FindFirst} locates the first occurrence, \texttt{FindNext} locates the next occurrence, and so on.

Each of the \texttt{Find} methods begins its search from the location and in the direction specified in the following table.

<table>
<thead>
<tr>
<th>Find method</th>
<th>Begins searching at</th>
<th>Search direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{FindFirst}</td>
<td>Beginning of recordset</td>
<td>End of recordset</td>
</tr>
<tr>
<td>\texttt{FindLast}</td>
<td>End of recordset</td>
<td>Beginning of recordset</td>
</tr>
<tr>
<td>\texttt{FindNext}</td>
<td>Current record</td>
<td>End of recordset</td>
</tr>
<tr>
<td>\texttt{FindPrevious}</td>
<td>Current record</td>
<td>Beginning of recordset</td>
</tr>
</tbody>
</table>

When you use the \texttt{FindLast} method, the \texttt{Microsoft Jet database engine} fully populates your \texttt{Recordset} before beginning the search, if this hasn't already happened.

Using one of the \texttt{Find} methods isn't the same as using a \texttt{Move} method, however, which simply makes the first, last, next, or previous record current without specifying a condition. You can follow a Find operation with a Move operation.
Always check the value of the **NoMatch** property to determine whether the Find operation has succeeded. If the search succeeds, **NoMatch** is **False**. If it fails, **NoMatch** is **True** and the current record isn't defined. In this case, you must position the current record pointer back to a valid record.

Using the **Find** methods with **Microsoft Jet-connected ODBC**-accessed recordsets can be inefficient. You may find that rephrasing your criteria to locate a specific record is faster, especially when working with large recordsets.

In an **ODBCDirect workspace**, the **Find** and **Seek** methods are not available on any type of **Recordset** object, because executing a **Find** or **Seek** through an ODBC connection is not very efficient over the network. Instead, you should design the query (that is, using the source argument to the **OpenRecordset** method) with an appropriate WHERE clause that restricts the returned records to only those that meet the criteria you would otherwise use in a **Find** or **Seek** method.

When working with Microsoft Jet-connected ODBC databases and large dynaset-type **Recordset** objects, you might discover that using the **Find** methods or using the **Sort** or **Filter** property is slow. To improve performance, use SQL queries with customized ORDER BY or WHERE clauses, **parameter queries**, or **QueryDef** objects that retrieve specific indexed records.

You should use the U.S. date format (month-day-year) when you search for fields containing dates, even if you're not using the U.S. version of the Microsoft Jet database engine; otherwise, the data may not be found. Use the Visual Basic **Format** function to convert the date. For example:

```vba
rstEmployees.FindFirst "HireDate > " & Format(mydate, 'm-d-yyyy') & "#"
```

If **criteria** is composed of a string concatenated with a non-integer value, and the system parameters specify a non-U.S. decimal character such as a comma (for example, `strSQL = "PRICE > " & lngPrice, and lngPrice = 125,50`), an error occurs when you try to call the method. This is because during concatenation, the number will be converted to a string using your system's default decimal character, and Microsoft Jet SQL only accepts U.S. decimal characters.

**Notes**
- For best performance, the criteria should be in either the form "field = value" where field is an indexed field in the underlying base table, or "field LIKE prefix" where field is an indexed field in the underlying base table and prefix is a prefix search string (for example, "ART*").

- In general, for equivalent types of searches, the Seek method provides better performance than the Find methods. This assumes that table-type Recordset objects alone can satisfy your needs.
GetChunk Method

Returns all or a portion of the contents of a Memo or Long Binary Field object in the Fields collection of a Recordset object.

Syntax

Set variable = recordset ! field.GetChunk (offset, numbytes)

The GetChunk method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>A Variant (String subtype) that receives the data from the Field object named by field.</td>
</tr>
<tr>
<td>recordset</td>
<td>An object variable that represents the Recordset object containing the Fields collection.</td>
</tr>
<tr>
<td>field</td>
<td>An object variable that represents a Field object whose Type property is set to dbMemo (Memo) or dbLongBinary (Long Binary).</td>
</tr>
<tr>
<td>offset</td>
<td>A Long value equal to the number of bytes to skip before copying begins.</td>
</tr>
</tbody>
</table>
numbytes: A Long value equal to the number of bytes you want to return.

Remarks

The bytes returned by GetChunk are assigned to variable. Use GetChunk to return a portion of the total data value at a time. You can use the AppendChunk method to reassemble the pieces.

If offset is 0, GetChunk begins copying from the first byte of the field.

If numbytes is greater than the number of bytes in the field, GetChunk returns the actual number of remaining bytes in the field.

Caution: Use a Memo field for text, and put binary data only in Long Binary fields. Doing otherwise will cause undesirable results.
GetRows Method

Retrieves multiple rows from a `Recordset` object.

**Syntax**

```vbscript
Set varArray = recordset.GetRows (numrows)
```

The `GetRows` method syntax has the following parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>varArray</code></td>
<td>A <code>Variant</code> that stores the returned data.</td>
</tr>
<tr>
<td><code>recordset</code></td>
<td>An <code>object variable</code> that represents a <code>Recordset</code> object.</td>
</tr>
<tr>
<td><code>numrows</code></td>
<td>A <code>Variant</code> that is equal to the number of rows to retrieve.</td>
</tr>
</tbody>
</table>

**Remarks**

Use the `GetRows` method to copy records from a `Recordset`. `GetRows` returns a two-dimensional `array`. The first subscript identifies the field and the second identifies the row number. For example, `intField` represents the field, and `intRecord` identifies the row number:
avarRecords(intField, intRecord)

To get the first field value in the second row returned, use code like the following:

field1 = avarRecords(0,1)

To get the second field value in the first row, use code like the following:

field2 = avarRecords(1,0)

The avarRecords variable automatically becomes a two-dimensional array when GetRows returns data.

If you request more rows than are available, then GetRows returns only the number of available rows. You can use the Visual Basic for Applications UBound function to determine how many rows GetRows actually retrieved, because the array is sized to fit the number of returned rows. For example, if you returned the results into a Variant called varA, you could use the following code to determine how many rows were actually returned:

numReturned = UBound(varA,2) + 1

You need to use "+ 1" because the first row returned is in the 0 element of the array. The number of rows that you can retrieve is constrained by the amount of available memory. You shouldn't use GetRows to retrieve an entire table into an array if it is large.

Because GetRows returns all fields of the Recordset into the array, including Memo and Long Binary fields, you might want to use a query that restricts the fields returned.

After you call GetRows, the current record is positioned at the next unread row. That is, GetRows has the same effect on the current record as Move numrows.

If you are trying to retrieve all the rows by using multiple GetRows calls, use the EOF property to be sure that you're at the end of the Recordset. GetRows returns less than the number requested if it's at the end of the Recordset, or if it can't retrieve a row in the range requested. For example, if you're trying to retrieve 10 records, but you can't retrieve the fifth record, GetRows returns four records and makes the fifth record the current record. This will not generate a
run-time error. This might occur if another user deletes a record in a dynaset-type Recordset. See the example for a demonstration of how to handle this.
Idle Method

Suspends data processing, enabling the Microsoft Jet database engine to complete any pending tasks, such as memory optimization or page timeouts (Microsoft Jet workspaces only).

Syntax

```
DBEngine.Idle [dbRefreshCache]
```

Remarks

The Idle method allows the Microsoft Jet database engine to perform background tasks that may not be up-to-date because of intense data processing. This is often true in multiuser, multitasking environments that don't have enough background processing time to keep all records in a Recordset current.

Usually, read locks are removed and data in local dynaset-type Recordset objects are updated only when no other actions (including mouse movements) occur. If you periodically use the Idle method, Microsoft Jet can catch up on background processing tasks by releasing unneeded read locks.

Specifying the optional dbRefreshCache argument refreshes memory with only
the most current data from the .mdb file. The `dbForceOSFlush` argument forces pending writes to .mdb files as part of the `CommitTrans` method.

You don't need to use this method in single-user environments unless multiple instances of an application are running. The `Idle` method may increase performance in a multiuser environment because it forces the database engine to write data to disk, releasing locks on memory.

**Note** You can also release read locks by making operations part of a transaction.
MakeReplica Method

Makes a new replica from another database replica (Microsoft Jet workspaces only).

Syntax

database.MakeReplica replica, description, options

The MakeReplica method syntax has the following parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>An object variable that represents an existing Database that is a replica.</td>
</tr>
<tr>
<td>replica</td>
<td>A String that is the path and file name of the new replica. If replica is an existing file name, then an error occurs.</td>
</tr>
<tr>
<td>description</td>
<td>A String that describes the replica that you are creating. Optional. A constant or combination of constants that specifies characteristics of the replica you are creating, as specified in Settings.</td>
</tr>
</tbody>
</table>

Settings
You can use one or more of the following constants in the `options` argument.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbRepMakePartial</code></td>
<td>Creates a <a href="#">partial replica</a>. Prevents users from modifying the replicable objects of the new replica; however, when you synchronize the new replica with another member of the replica set, design and data changes will be propagated to the new replica.</td>
</tr>
<tr>
<td><code>dbRepMakeReadOnly</code></td>
<td>A newly created partial replica will have all <a href="#">ReplicaFilter</a> properties set to <code>False</code>, meaning that no data will be in the tables.</td>
</tr>
</tbody>
</table>
Move Method

Moves the position of the current record in a Recordset object.

Syntax

recordset.Move rows, start

The Move method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>An object variable that represents the Recordset object whose current record position is being moved.</td>
</tr>
<tr>
<td>rows</td>
<td>A signed Long value specifying the number of rows the position will move. If rows is greater than 0, the position is moved forward (toward the end of the file). If rows is less than 0, the position is moved backward (toward the beginning of the file).</td>
</tr>
<tr>
<td>startbookmark</td>
<td>Optional. A Variant (String subtype) value identifying a bookmark. If you specify startbookmark, the move begins relative to this bookmark. Otherwise, Move begins from the current record.</td>
</tr>
</tbody>
</table>
Remarks

If you use Move to position the current record pointer before the first record, the current record pointer moves to the beginning of the file. If the Recordset contains no records and its BOF property is True, using this method to move backward causes an error.

If you use Move to position the current record pointer after the last record, the current record pointer position moves to the end of the file. If the Recordset contains no records and its EOF property is True, then using this method to move forward causes an error.

If either the BOF or EOF property is True and you attempt to use the Move method without a valid bookmark, a run-time error occurs.

Notes

- When you use Move on a forward-only-type Recordset object, the rows argument must be a positive integer and bookmarks aren't allowed. This means you can only move forward.

- To make the first, last, next, or previous record in a Recordset the current record, use either the MoveFirst, MoveLast, MoveNext, or MovePrevious method.

- Using Move with rows equal to 0 is an easy way to retrieve the underlying data for the current record. This is useful if you want to make sure that the current record has the most recent data from the base tables. It will also cancel any pending Edit or AddNew calls.
MoveFirst, MoveLast, MoveNext, MovePrevious Methods

Move to the first, last, next, or previous record in a specified Recordset object and make that record the current record.

Syntax

```
recordset.{MoveFirst | MoveLast [dbRunAsync] | MoveNext | MovePrevious}
```

The recordset placeholder is an object variable that represents an open Recordset object.

Remarks

Use the Move methods to move from record to record without applying a condition.

Caution  If you edit the current record, be sure you use the Update method to save the changes before you move to another record. If you move to another record without updating, your changes are lost without warning.
When you open a **Recordset**, the first record is current and the **BOF** property is **False**. If the **Recordset** contains no records, the **BOF** property is **True**, and there is no current record.

If the first or last record is already current when you use **MoveFirst** or **MoveLast**, the current record doesn't change.

If you use **MovePrevious** when the first record is current, the **BOF** property is **True**, and there is no current record. If you use **MovePrevious** again, an error occurs, and **BOF** remains **True**.

If you use **MoveNext** when the last record is current, the **EOF** property is **True**, and there is no current record. If you use **MoveNext** again, an error occurs, and **EOF** remains **True**.

If **recordset** refers to a table-type **Recordset** (**Microsoft Jet workspaces** only), movement follows the **current index**. You can set the current index by using the **Index** property. If you don't set the current index, the order of returned records is undefined.

**Important**  You can use the **MoveLast** method to fully populate a dynaset- or snapshot-type **Recordset** to provide the current number of records in the **Recordset**. However, if you use **MoveLast** in this way, you can slow down your application's performance. You should only use **MoveLast** to get a record count if it is absolutely necessary to obtain an accurate record count on a newly opened **Recordset**. If you use the **dbRunAsync** constant with **MoveLast**, the method call is asynchronous. You can use the **StillExecuting** property to determine when the **Recordset** is fully populated, and you can use the **Cancel** method to terminate execution of the asynchronous **MoveLast** method call.

You can't use the **MoveFirst**, **MoveLast**, and **MovePrevious** methods on a **forward-only–type** **Recordset** object.

To move the position of the current record in a **Recordset** object a specific number of records forward or backward, use the **Move** method.
NewPassword Method

Changes the password of an existing user account or Microsoft Jet database (Microsoft Jet workspaces only).

Syntax

```
object.NewPassword oldpassword, newpassword
```

The NewPassword method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>An object variable that represents the User object or a Microsoft Jet 3.x Database object whose Password property you want to change.</td>
</tr>
<tr>
<td>oldpassword</td>
<td>A String that is the current setting of the Password property of the User or Jet 3.x Database object.</td>
</tr>
<tr>
<td>newpassword</td>
<td>A String that is the new setting of the Password property of the User or Jet 3.x Database object.</td>
</tr>
</tbody>
</table>

Remarks
The *oldpassword* and *newpassword* strings can be up to 14 characters long and can include any characters except the ASCII character 0 (null). To clear the password, use a zero-length string (""") for *newpassword*.

Passwords are case-sensitive.

If *object* refers to a **User** object that is not yet appended to a **Users** collection, an error occurs. To set a new password, you must either log on as the user whose account you're changing, or you must be a member of the Admins group. The **Password** property of a **User** object is write-only — users can't read the current value.

If *object* refers to a Microsoft Jet version 3.0 or later **Database** object, this method offers some security by means of password protection. When you create or open a Microsoft Jet 3.x .mdb file, part of the **Connect connection string** can describe the password.

If a database has no password, Microsoft Jet will automatically create one by passing a zero-length string (""") for the old password.

**Caution** If you lose your password, you can never open the database again.
NextRecordset Method

Gets the next set of records, if any, returned by a multi-part select query in an OpenRecordset call, and returns a Boolean value indicating whether one or more additional records are pending (ODBCDirect workspaces only).

Syntax

boolean = recordset.NextRecordset

The NextRecordset method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>A Boolean variable. True indicates the next set of records is available in recordset; False indicates that no more records are pending and recordset is now empty.</td>
</tr>
<tr>
<td>recordset</td>
<td>An existing Recordset variable to which you want to return pending records.</td>
</tr>
</tbody>
</table>

Remarks
In an ODBCDirect workspace, you can open a Recordset containing more than one select query in the source argument of OpenRecordset, or the SQL property of a select query QueryDef object, as in the following example.

SELECT LastName, FirstName FROM Authors
WHERE LastName = 'Smith';
SELECT Title, ISBN FROM Titles
WHERE Pub_ID = 9999

The returned Recordset will open with the results of the first query. To obtain the result sets of records from subsequent queries, use the NextRecordset method.

If more records are available (that is, there was another select query in the OpenRecordset call or in the SQL property), the records returned from the next query will be loaded into the Recordset, and NextRecordset will return True, indicating that the records are available. When no more records are available (that is, results of the last select query have been loaded into the Recordset), then NextRecordset will return False, and the Recordset will be empty.

You can also use the Cancel method to flush the contents of a Recordset. However, Cancel also flushes any additional records not yet loaded.
OpenConnection Method

Opens a Connection object on an ODBC data source (ODBCDirect workspaces only).

Syntax

Set connection = workspace.OpenConnection (name, options, readonly, connect)

The OpenConnection method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>A Connection object variable to which the new connection will be assigned. Optional. A variable of a Workspace object data type that references the existing Workspace object that will contain the new connection.</td>
</tr>
<tr>
<td>workspace</td>
<td>A string expression. See the discussion under Remarks. Optional. A Variant that sets various options for the connection, as specified in Settings. Based on this</td>
</tr>
</tbody>
</table>
The **options** argument determines if and when to prompt the user to establish the connection, and whether or not to open the connection asynchronously. You can use one of the following constants.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbDriverNoPrompt</td>
<td>The ODBC Driver Manager uses the connection string provided in dbname and connect. If you don't provide sufficient information, a run-time error occurs.</td>
</tr>
<tr>
<td>dbDriverPrompt</td>
<td>Default. If the connect argument includes all the necessary information to complete a connection, the ODBC Driver Manager uses the string in connect. Otherwise it behaves as it does when you specify.</td>
</tr>
<tr>
<td>dbDriverComplete</td>
<td>The ODBC Data Sources dialog box, which displays any relevant information supplied in dbname or connect. The connection string is made up of the DSN that the user selects via the dialog boxes, or, if the user doesn't specify a DSN, the default DSN is used.</td>
</tr>
</tbody>
</table>

**Settings**

The **options** argument determines if and when to prompt the user for connection information such as data source name (DSN), user name, and password. Optional. A **Boolean** value that is **True** if the connection is to be opened for read-only access and **False** if the connection is to be opened for read/write access (default).

Optional. An ODBC connect string. See the **Connect** property for the specific elements and syntax of this string. A prepended "ODBC:" is required. If connect is omitted, the UID and/or PWD will be taken from the **UserName** and **Password** properties of the **Workspace**.
**dbDriverPrompt.**
This option behaves like **dbDriverComplete** except the **ODBC driver** disables the prompts for any information not required to complete the connection.

**dbDriverCompleteRequired**
Execute the method asynchronously. This constant may be used with any of the other **options** constants.

**Remarks**

Use the **OpenConnection** method to establish a connection to an ODBC data source from an ODBCDirect workspace. The **OpenConnection** method is similar but not equivalent to **OpenDatabase**. The main difference is that **OpenConnection** is available only in an ODBCDirect workspace.

If you specify a registered ODBC data source name (DSN) in the **connect** argument, then the **name** argument can be any valid string, and will also provide the **Name** property for the **Connection** object. If a valid DSN is not included in the **connect** argument, then **name** must refer to a valid ODBC DSN, which will also be the **Name** property. If neither **name** nor **connect** contains a valid DSN, the ODBC driver manager can be set (via the **options** argument) to prompt the user for the required connection information. The DSN supplied through the prompt then provides the **Name** property.

**OpenConnection** returns a **Connection** object which contains information about the connection. The **Connection** object is similar to a **Database** object. The principal difference is that a **Database** object usually represents a database, although it can be used to represent a connection to an ODBC data source from a **Microsoft Jet workspace**.
OpenDatabase Method

Opens a specified database in a *Workspace* object and returns a reference to the *Database* object that represents it.

**Syntax**

```vba
Set database = workspace.OpenDatabase (dbname, options, read-only, connect)
```

The *OpenDatabase* method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>database</code></td>
<td>An object variable that represents the <em>Database</em> object that you want to open. Optional. An object variable that represents the existing <em>Workspace</em> object that will contain the database. If you don't include a value for <code>workspace</code>, <em>OpenDatabase</em> uses the default workspace.</td>
</tr>
<tr>
<td><code>dbname</code></td>
<td>A String that is the name of an existing <em>Microsoft Jet database</em> file, or the data source name (DSN) of an <em>ODBC data source</em>. See the <em>Name</em> property for more information</td>
</tr>
</tbody>
</table>
about setting this value.

$options$

Optional. A Variant that sets various options for the database, as specified in Settings.

Optional. A Variant (Boolean subtype) value that is True if you want to open the database with read-only access, or False (default) if you want to open the database with read/write access.

$read-only$

Optional. A Variant (String subtype) that specifies various connection information, including passwords.

$connect$

Settings

For Microsoft Jet workspaces, you can use the following values for the options argument.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>Opens the database in exclusive mode.</td>
</tr>
<tr>
<td></td>
<td>(Default) Opens the database in shared mode.</td>
</tr>
</tbody>
</table>

For ODBCDirect workspaces, the options argument determines if and when to prompt the user to establish the connection. You can use one of the following constants.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbDriverNoPrompt</td>
<td>The ODBC Driver Manager uses the connection string provided in dbname and connect. If you don't provide sufficient information, a run-time error occurs. The ODBC Driver Manager displays the ODBC Data Sources dialog box, which displays any relevant information supplied in dbname or connect. The connection string is made up of the DSN that the user selects via the dialog boxes, or, if the user doesn't specify a DSN, the default DSN is used.</td>
</tr>
<tr>
<td>dbDriverPrompt</td>
<td></td>
</tr>
</tbody>
</table>


**dbDriverComplete**

(Default) If the `connect` and `dbname` arguments include all the necessary information to complete a connection, the ODBC Driver Manager uses the string in `connect`. Otherwise it behaves as it does when you specify `dbDriverPrompt`. This option behaves like `dbDriverCompleteRequired` except the ODBC driver disables the prompts for any information not required to complete the connection.

**dbDriverCompleteRequired**

This option behaves like `dbDriverComplete` except the ODBC driver disables the prompts for any information not required to complete the connection.

**Remarks**

When you open a database, it is automatically added to the Databases collection. Further, in an ODBCDirect workspace, the Connection object corresponding to the new Database object is also created and appended to the Connections collection of the same Workspace object.

Some considerations apply when you use `dbname`:

- If it refers to a database that is already open for exclusive access by another user, an error occurs.
- If it doesn't refer to an existing database or valid ODBC data source name, an error occurs.
- If it's a zero-length string ("") and `connect` is "ODBC;", a dialog box listing all registered ODBC data source names is displayed so the user can select a database.
- If you're opening a database through an ODBCDirect workspace and you provide the DSN in `connect`, you can set `dbname` to a string of your choice that you can use to reference this database in subsequent code.

The `connect` argument is expressed in two parts: the database type, followed by a semicolon (;) and the optional arguments. You must first provide the database type, such as "ODBC;" or "Excel 5.0;". The optional arguments follow in no
particular order, separated by semicolons. One of the parameters may be the password (if one is assigned). For example:

"Excel 5.0; pwd=mypassword"

Using the NewPassword method on a Database object other than an ODBCDirect database changes the password parameter that appears in the ";pwd=..." part of this argument. You must supply the options and read-only arguments to supply a source string. See the Connect property for syntax.

To close a database, and thus remove the Database object from the Databases collection, use the Close method on the object.

**Note** When you access a Microsoft Jet-connected ODBC data source, you can improve your application's performance by opening a Database object connected to the ODBC data source, rather than by linking individual TableDef objects to specific tables in the ODBC data source.
OpenRecordset Method

Creates a new Recordset object and appends it to the Recordsets collection.

Syntax

For Connection and Database objects:

Set recordset = object.OpenRecordset (source, type, options, lockedits)

For QueryDef, Recordset, and TableDef objects:

Set recordset = object.OpenRecordset (type, options, lockedits)

The OpenRecordset method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>An object variable that represents the Recordset object you want to open.</td>
</tr>
<tr>
<td>object</td>
<td>An object variable that represents an existing object from which you want to create the new Recordset.</td>
</tr>
<tr>
<td></td>
<td>A String specifying the source of the records for the new Recordset.</td>
</tr>
</tbody>
</table>
source **Recordset**. The source can be a table name, a query name, or an **SQL statement** that returns records. For table-type **Recordset** objects in **Microsoft Jet databases**, the source can only be a table name.

type Optional. A constant that indicates the type of **Recordset** to open, as specified in Settings.

options Optional. A combination of constants that specify characteristics of the new **Recordset**, as listed in Settings.

lockedits Optional. A constant that determines the locking for the **Recordset**, as specified in Settings.

**Settings**

You can use one of the following constants for the **type** argument.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbOpenTable</td>
<td>Opens a table-type <strong>Recordset</strong> object (Microsoft Jet workspaces only).</td>
</tr>
<tr>
<td></td>
<td>Opens a dynamic-type <strong>Recordset</strong> object, which is similar to an ODBC dynamic cursor. (ODBCDirect workspaces only)</td>
</tr>
<tr>
<td>dbOpenDynaset</td>
<td>Opens a dynaset-type <strong>Recordset</strong> object, which is similar to an ODBC keyset cursor.</td>
</tr>
<tr>
<td>dbOpenSnapshot</td>
<td>Opens a snapshot-type <strong>Recordset</strong> object, which is similar to an ODBC static cursor.</td>
</tr>
<tr>
<td>dbOpenForwardOnly</td>
<td>Opens a forward-only-type <strong>Recordset</strong> object.</td>
</tr>
</tbody>
</table>

**Note** If you open a **Recordset** in a Microsoft Jet workspace and you don't specify a type, **OpenRecordset** creates a table-type **Recordset**, if possible. If you specify a linked table or query, **OpenRecordset** creates a dynaset-type **Recordset**. In an ODBCDirect workspace, the default setting is **dbOpenForwardOnly**.

You can use a combination of the following constants for the **options** argument.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>


dbAppendOnly

Allows users to append new records to the Recordset, but prevents them from editing or deleting existing records (Microsoft Jet dynaset-type Recordset only).

dbSQLPassThrough

Passes an SQL statement to a Microsoft Jet-connected ODBC data source for processing (Microsoft Jet snapshot-type Recordset only).

Generates a run-time error if one user is changing data that another user is editing (Microsoft Jet dynaset-type Recordset only). This is useful in applications where multiple users have simultaneous read/write access to the same data.

dbSeeChanges

Prevents other users from modifying or adding records (Microsoft Jet Recordset objects only).

dbDenyRead

Prevents other users from reading data in a table (Microsoft Jet table-type Recordset only).

Creates a forward-only Recordset (Microsoft Jet snapshot-type Recordset only). It is provided only for backward compatibility, and you should use the dbOpenForwardOnly constant in the type argument instead of using this option.

Prevents users from making changes to the Recordset (Microsoft Jet only). The dbReadOnly constant in the lockedits argument replaces this option, which is provided only for backward compatibility.

dbRunAsync

Runs an asynchronous query (ODBCDirect workspaces only).

Runs a query by skipping SQLPrepare and directly calling SQLEXecDirect (ODBCDirect workspaces only). Use this option only when you’re not opening a Recordset based on a parameter query. For more information, see the "Microsoft ODBC 3.0 Programmer’s Reference."
**dbInconsistent**  Allows inconsistent updates (Microsoft Jet dynaset-type and snapshot-type Recordset objects only).

**dbConsistent**  Allows only consistent updates (Microsoft Jet dynaset-type and snapshot-type Recordset objects only).

**Note**  The constants **dbConsistent** and **dbInconsistent** are mutually exclusive, and using both causes an error. Supplying a lockedits argument when options uses the **dbReadOnly** constant also causes an error.

You can use the following constants for the lockedits argument.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dbReadOnly</strong></td>
<td>Prevents users from making changes to the Recordset (default for ODBCDirect workspaces). You can use <strong>dbReadOnly</strong> in either the options argument or the lockedits argument, but not both. If you use it for both arguments, a run-time error occurs.</td>
</tr>
<tr>
<td><strong>dbPessimistic</strong></td>
<td>Uses pessimistic locking to determine how changes are made to the Recordset in a multiuser environment. The page containing the record you're editing is locked as soon as you use the Edit method (default for Microsoft Jet workspaces). Uses optimistic locking to determine how changes are made to the Recordset in a multiuser environment. The page containing the record is not locked until the Update method is executed.</td>
</tr>
<tr>
<td><strong>dbOptimistic</strong></td>
<td>Uses optimistic concurrency based on row values (ODBCDirect workspaces only).</td>
</tr>
<tr>
<td><strong>dbOptimisticValue</strong></td>
<td>Enables batch optimistic updating (ODBCDirect workspaces only).</td>
</tr>
<tr>
<td><strong>dbOptimisticBatch</strong></td>
<td></td>
</tr>
</tbody>
</table>
Remarks

In a Microsoft Jet workspace, if object refers to a QueryDef object, or a dynaset- or snapshot-type Recordset, or if source refers to an SQL statement or a TableDef that represents a linked table, you can't use dbOpenTable for the type argument; if you do, a run-time error occurs. If you want to use an SQL pass-through query on a linked table in a Microsoft Jet-connected ODBC data source, you must first set the Connect property of the linked table's database to a valid ODBC connection string. If you only need to make a single pass through a Recordset opened from a Microsoft Jet-connected ODBC data source, you can improve performance by using dbOpenForwardOnly for the type argument.

If object refers to a dynaset- or snapshot-type Recordset, the new Recordset is of the same type object. If object refers to a table-type Recordset object, the type of the new object is a dynaset-type Recordset. You can't open new Recordset objects from forward-only–type or ODBCDirect Recordset objects.

In an ODBCDirect workspace, you can open a Recordset containing more than one select query in the source argument, such as

"SELECT LastName, FirstName FROM Authors
WHERE LastName = 'Smith';
SELECT Title, ISBN FROM Titles
WHERE ISBN Like '1-55615-*'"

The returned Recordset will open with the results of the first query. To obtain the result sets of records from subsequent queries, use the NextRecordset method.

Note  You can send DAO queries to a variety of different database servers with ODBCDirect, and different servers will recognize slightly different dialects of SQL. Therefore, context-sensitive Help is no longer provided for Microsoft Jet SQL, although online Help for Microsoft Jet SQL is still included through the Help menu. Be sure to check the appropriate reference documentation for the SQL dialect of your database server when using either ODBCDirect connections or pass-through queries in Microsoft Jet-connected client/server applications.

Use the dbSeeChanges constant in a Microsoft Jet workspace if you want to trap changes while two or more users are editing or deleting the same record. For example, if two users start editing the same record, the first user to execute the
Update method succeeds. When the second user invokes the Update method, a run-time error occurs. Similarly, if the second user tries to use the Delete method to delete the record, and the first user has already changed it, a run-time error occurs.

Typically, if the user gets this error while updating a record, your code should refresh the contents of the fields and retrieve the newly modified values. If the error occurs while deleting a record, your code could display the new record data to the user and a message indicating that the data has recently changed. At this point, your code can request a confirmation that the user still wants to delete the record.

You should also use the dbSeeChanges constant if you open a Recordset in a Microsoft Jet-connected ODBC workspace against a Microsoft SQL Server 6.0 (or later) table that has an IDENTITY column, otherwise an error may result.

In an ODBCDirect workspace, you can execute asynchronous queries by setting the dbRunAsync constant in the options argument. This allows your application to continue processing other statements while the query runs in the background. But, you cannot access the Recordset data until the query has completed. To determine whether the query has finished executing, check the StillExecuting property of the new Recordset. If the query takes longer to complete than you anticipated, you can terminate execution of the query with the Cancel method.

Opening more than one Recordset on an ODBC data source may fail because the connection is busy with a prior OpenRecordset call. One way around this is to use a server-side cursor and ODBCDirect, if the server supports this. Another solution is to fully populate the Recordset by using the MoveLast method as soon as the Recordset is opened.

If you open a Connection object with DefaultCursorDriver set to dbUseClientBatchCursor, you can open a Recordset to cache changes to the data (known as batch updating) in an ODBCDirect workspace. Include dbOptimisticBatch in the lockedits argument to enable update caching. See the Update method topic for details about how to write changes to disk immediately, or to cache changes and write them to disk as a batch.

Closing a Recordset with the Close method automatically deletes it from the Recordsets collection.
**Note** If *source* refers to an SQL statement composed of a string concatenated with a non-integer value, and the system parameters specify a non-U.S. decimal character such as a comma (for example, `strSQL = "PRICE > " & lngPrice, and `lngPrice = 125,50`), an error occurs when you try to open the **Recordset**. This is because during concatenation, the number will be converted to a string using your system’s default decimal character, and SQL only accepts U.S. decimal characters.
PopulatePartial Method

Synchronizes any changes in a partial replica with the full replica, clears all records in the partial replica, and then repopulates the partial replica based on the current replica filters. (Microsoft Jet databases only.)

Syntax

`database.PopulatePartial dbname`

The PopulatePartial method syntax has the following parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>An object variable that references the partial replica Database object that you want to populate.</td>
</tr>
<tr>
<td>dbname</td>
<td>A string specifying the path and name of the full replica from which to populate records.</td>
</tr>
</tbody>
</table>

Remarks

When you synchronize a partial replica with a full replica, it is possible to create "orphaned" records in the partial replica. For example, suppose you have a
Customers table with its ReplicaFilter set to "Region = 'CA'". If a user changes a customer's region from CA to NY in the partial replica, and then a synchronization occurs via the Synchronize method, the change is propagated to the full replica but the record containing NY in the partial replica is orphaned because it now doesn't meet the replica filter criteria.

To solve the problem of orphaned records, you can use the PopulatePartial method. The PopulatePartial method is similar to the Synchronize method, but it synchronizes any changes with the full replica, removes all records in the partial replica, and then repopulates the partial replica based on the current replica filters. Even if your replica filters have not changed, PopulatePartial will always clear all records in the partial replica and repopulate it based on the current filters.

Generally, you should use the PopulatePartial method when you create a partial replica and whenever you change your replica filters. If your application changes replica filters, you should follow these steps:

1. Synchronize your full replica with the partial replica in which the filters are being changed.

2. Use the ReplicaFilter and PartialReplica properties to make the desired changes to the replica filter.

3. Call the PopulatePartial method to remove all records from the partial replica and transfer all records from the full replica that meet the new replica filter criteria.

If a replica filter has changed, and the Synchronize method is invoked without first invoking PopulatePartial, a trappable error occurs.

The PopulatePartial method can only be invoked on a partial replica that has been opened for exclusive access. Furthermore, you can't call the PopulatePartial method from code running within the partial replica itself. Instead, open the partial replica exclusively from the full replica or another database, then call PopulatePartial.

Note   Although PopulatePartial performs a one-way synchronization before clearing and repopulating the partial replica, it is still a good idea to call
**Synchronize** before calling **PopulatePartial**. This is because if the call to **Synchronize** fails, a trappable error occurs. You can use this error to decide whether or not to proceed with the **PopulatePartial** method (which removes all records in the partial replica). If **PopulatePartial** is called by itself and an error occurs while records are being synchronized, records in the partial replica will still be cleared, which may not be the desired result.
Refresh Method

Updates the objects in a collection to reflect the current database's schema.

Syntax

collection.Refresh

The collection placeholder is an object variable that represents a persistent collection.

Remarks

You can't use the Refresh method with collections that aren't persistent, such as Connections, Databases, Recordsets, Workspaces, or the QueryDefs collection of a Connection object.

To determine the position that the Microsoft Jet database engine uses for Field objects in the Fields collection of a QueryDef, Recordset, or TableDef object, use the OrdinalPosition property of each Field object. Changing the OrdinalPosition property of a Field object may not change the order of the Field objects in the collection until you use the Refresh method.
Use the **Refresh** method in multiuser environments in which other users may change the database. You may also need to use it on any collections that are indirectly affected by changes to the database. For example, if you change a **Users** collection, you may need to refresh a **Groups** collection before using the **Groups** collection.

A collection is filled with objects the first time it's referred to and won't automatically reflect subsequent changes other users make. If it's likely that another user has changed a collection, use the **Refresh** method on the collection immediately before carrying out any task in your application that assumes the presence or absence of a particular object in the collection. This will ensure that the collection is as up-to-date as possible. On the other hand, using **Refresh** can unnecessarily slow performance.
RefreshLink Method

Updates the connection information for a linked table (Microsoft Jet workspaces only).

Syntax

tabledef.RefreshLink

The tabledef placeholder specifies the TableDef object representing the linked table whose connection information you want to update.

Remarks

To change the connection information for a linked table, reset the Connect property of the corresponding TableDef object and then use the RefreshLink method to update the information. Using RefreshLink method doesn't change the linked table's properties and Relation objects.

For this connection information to exist in all collections associated with the TableDef object that represents the linked table, you must use the Refresh method on each collection.
RegisterDatabase Method

Enters connection information for an ODBC data source in the Windows Registry. The ODBC driver needs connection information when the ODBC data source is opened during a session.

Syntax

DBEngine/RegisterDatabase dbname, driver, silent, attributes

The RegisterDatabase method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbname</td>
<td>A String that is the name used in the OpenDatabase method. It refers to a block of descriptive information about the data source. For example, if the data source is an ODBC remote database, it could be the name of the server. A String that is the name of the ODBC driver. This isn't the name of the ODBC driver DLL file. For example, SQL Server is a driver name, but SQLSRVR.dll is the name of a DLL file. You must have ODBC and the appropriate driver already installed.</td>
</tr>
</tbody>
</table>
A **Boolean** that is **True** if you don't want to display the ODBC driver dialog boxes that prompt for driver-specific information; or **False** if you want to display the ODBC driver dialog boxes. If **silent** is **True**, **attributes** must contain all the necessary driver-specific information or the dialog boxes are displayed anyway.

**attributes**  
A **String** that is a list of keywords to be added to the Windows Registry. The keywords are in a carriage-return–delimited string.

**Remarks**

If the database is already registered (connection information is already entered) in the Windows Registry when you use the `RegisterDatabase` method, the connection information is updated.

If the `RegisterDatabase` method fails for any reason, no changes are made to the Windows Registry, and an error occurs.

For more information about ODBC drivers such as SQL Server, see the Help file provided with the driver.

You should use the **ODBC Data Sources** dialog box in the Control Panel to add new data sources, or to make changes to existing entries. However, if you use the `RegisterDatabase` method, you should set the **silent** option to **True**.
RepairDatabase Method

Note  The RepairDatabase method is no longer available, but listed for backwards compatibility. Use the CompactDatabase method instead.

Attempts to repair a corrupted Microsoft Jet database (Microsoft Jet databases only).

Syntax

DBEngine.RepairDatabase dbname

The dbname argument is a String that is the path and file name for an existing Microsoft Jet database file. If you omit the path, only the current directory is searched. If your system supports the uniform naming convention (UNC), you can also specify a network path, such as "\\server1\share1\dir1\db1.mdb".

Remarks

You must close the database specified by dbname before you repair it. In a multiuser environment, other users can't have dbname open while you're repairing it. If dbname isn't closed or isn't available for exclusive use, an error occurs.
This method attempts to repair a database that was marked as possibly corrupt by an incomplete write operation. This can occur if an application using the Microsoft Jet database engine is closed unexpectedly because of a power outage or computer hardware problem. The database won't be marked as possibly corrupt if you use the Close method or if you quit your application in a usual way.

The RepairDatabase method also attempts to validate all system tables and all indexes. Any data that can't be repaired is discarded. If the database can't be repaired, a run-time error occurs.

When you attempt to open or compact a corrupted database, a run-time error usually occurs. In some situations, however, a corrupted database may not be detected, and no error occurs. It's a good idea to provide your users with a way to use the RepairDatabase method in your application if their database behaves unpredictably.

Some types of databases can become corrupted if a user ends an application without closing Database or Recordset objects and the Microsoft Jet database engine; Microsoft Windows doesn't have a chance to write data caches to disk. To avoid corrupt databases, establish procedures for closing applications and shutting down systems that ensure that all cached pages are saved to the database. In some cases, power supplies that can't be interrupted may be necessary to prevent accidental data loss during power fluctuations.

**Note** After repairing a database, it's also a good idea to compact it using the CompactDatabase method to defragment the file and to recover disk space.
Requery Method

Updates the data in a Recordset object by re-executing the query on which the object is based.

Syntax

recordset.Requery newquerydef

The Requery method syntax has the following parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>An object variable that represents an existing Microsoft Jet dynaset-, snapshot-, or forward-only–type Recordset object, or an ODBCDirect Recordset object.</td>
</tr>
<tr>
<td>newquerydef</td>
<td>Optional. A Variant that represents the Name property value of a QueryDef object (Microsoft Jet workspaces only).</td>
</tr>
</tbody>
</table>

Remarks

Use this method to make sure that a Recordset contains the most recent data.
This method re-populates the current **Recordset** by using either the current query parameters or (in a Microsoft Jet workspace) the new ones supplied by the `newquerydef` argument.

In an **ODBCDirect workspace**, if the original query was **asynchronous**, then **Requery** will also execute an asynchronous query.

If you don't specify a `newquerydef` argument, the **Recordset** is re-populated based on the same query definition and parameters used to originally populate the **Recordset**. Any changes to the underlying data will be reflected during this re-population. If you didn't use a **QueryDef** to create the **Recordset**, the **Recordset** is re-created from scratch.

If you specify the original **QueryDef** in the `newquerydef` argument, then the **Recordset** is queried using the parameters specified by the **QueryDef**. Any changes to the underlying data will be reflected during this re-population. To reflect any changes to the query parameter values in the **Recordset**, you must supply the `newquerydef` argument.

If you specify a different **QueryDef** than what was originally used to create the **Recordset**, the **Recordset** is re-created from scratch.

When you use **Requery**, the first record in the **Recordset** becomes the **current record**.

You can't use the **Requery** method on dynaset- or snapshot-type **Recordset** objects whose **Restartable** property is set to **False**. However, if you supply the optional `newquerydef` argument, the **Restartable** property is ignored.

If both the **BOF** and **EOF** property settings of the **Recordset** object are **True** after you use the **Requery** method, the query didn't return any records and the **Recordset** contains no data.
Seek Method

Locates the record in an indexed table-type `Recordset` object that satisfies the specified criteria for the current index and makes that record the current record (Microsoft Jet workspaces only).

**Syntax**

```
recordset.Seek comparison, key1, key2...key13
```

The **Seek** method syntax has the following parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recordset</code></td>
<td>An object variable that represents an existing table-type <code>Recordset</code> object that has a defined index as specified by the <code>Recordset</code> object's <code>Index</code> property.</td>
</tr>
<tr>
<td><code>comparison</code></td>
<td>One of the following string expressions: <code>&lt;</code>, <code>&lt;=</code>, <code>=</code>, <code>&gt;=</code>, or <code>&gt;</code>. One or more values corresponding to fields in the <code>Recordset</code> object's current index, as specified by its <code>Index</code> property setting. You can use up to 13 <code>key</code> arguments.</td>
</tr>
<tr>
<td><code>key1, key2...key13</code></td>
<td><code>Recordset</code> object's current index, as specified by its <code>Index</code> property setting. You can use up to 13 <code>key</code> arguments.</td>
</tr>
</tbody>
</table>

**Remarks**
You must set the current index with the **Index** property before you use **Seek**. If the index identifies a nonunique key field, **Seek** locates the first record that satisfies the criteria.

The **Seek** method searches through the specified key fields and locates the first record that satisfies the criteria specified by *comparison* and *key1*. Once found, it makes that record current and sets the **NoMatch** property to **False**. If the **Seek** method fails to locate a match, the **NoMatch** property is set to **True**, and the current record is undefined.

If *comparison* is equal (=), greater than or equal (>=), or greater than (>), **Seek** starts at the beginning of the index and searches forward.

If *comparison* is less than (<) or less than or equal (<=), **Seek** starts at the end of the index and searches backward. However, if there are duplicate index entries at the end of the index, **Seek** starts at an arbitrary entry among the duplicates and then searches backward.

You must specify values for all fields defined in the index. If you use **Seek** with a multiple-column index, and you don't specify a comparison value for every field in the index, then you cannot use the equal (=) operator in the comparison. That's because some of the criteria fields (*key2*, *key3*, and so on) will default to **Null**, which will probably not match. Therefore, the equal operator will work correctly only if you have a record which is all **Null** except the key you're looking for. It's recommended that you use the greater than or equal (>=) operator instead.

The *key1* argument must be of the same **field data type** as the corresponding field in the current index. For example, if the current index refers to a number field (such as Employee ID), *key1* must be numeric. Similarly, if the current index refers to a Text field (such as Last Name), *key1* must be a string.

There doesn't have to be a current record when you use **Seek**.

You can use the **Indexes** collection to enumerate the existing indexes.

To locate a record in a dynaset- or snapshot-type **Recordset** that satisfies a specific condition that is not covered by existing indexes, use the **Find** methods. To include all records, not just those that satisfy a specific condition, use the **Move** methods to move from record to record.
You can't use the **Seek** method on a [linked table](#) because you can't open linked tables as table-type **Recordset** objects. However, if you use the **OpenDatabase** method to directly open an **installable ISAM** (non-ODBC) database, you can use **Seek** on tables in that database.

In an **ODBCDirect workspace**, the **Find** and **Seek** methods are not available on any type of **Recordset** object, because executing a **Find** or **Seek** through an ODBC connection is not very efficient over the network. Instead, you should design the query (that is, using the **source** argument to the **OpenRecordset** method) with an appropriate WHERE clause that restricts the returned records to only those that meet the criteria you would otherwise use in a **Find** or **Seek**.
SetOption Method

Temporarily overrides values for the Microsoft Jet database engine keys in the Windows Registry (Microsoft Jet workspaces only).

Syntax

`DBEngine.SetOption parameter, newvalue`

The `SetOption` method syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parameter</code></td>
<td>A <code>Long</code> constant as described in Settings.</td>
</tr>
<tr>
<td><code>newvalue</code></td>
<td>A <code>Variant</code> value that you want to set <code>parameter</code> to.</td>
</tr>
</tbody>
</table>

Settings

Each constant refers to the corresponding registry key in the path `Jet\3.5\Engines\Jet 3.5\` (that is, `dbSharedAsyncDelay` corresponds to the key `Jet\3.5\Engines\Jet 3.5\SharedAsyncDelay`, and so on.).
<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbPageTimeout</td>
<td>The PageTimeout key</td>
</tr>
<tr>
<td>dbSharedAsyncDelay</td>
<td>The SharedAsyncDelay key</td>
</tr>
<tr>
<td>dbExclusiveAsyncDelay</td>
<td>The ExclusiveAsyncDelay key</td>
</tr>
<tr>
<td>dbLockRetry</td>
<td>The LockRetry key</td>
</tr>
<tr>
<td>dbUserCommitSync</td>
<td>The UserCommitSync key</td>
</tr>
<tr>
<td>dbImplicitCommitSync</td>
<td>The ImplicitCommitSync key</td>
</tr>
<tr>
<td>dbMaxBufferSize</td>
<td>The MaxBufferSize key</td>
</tr>
<tr>
<td>dbMaxLocksPerFile</td>
<td>The MaxLocksPerFile key</td>
</tr>
<tr>
<td>dbLockDelay</td>
<td>The LockDelay key</td>
</tr>
<tr>
<td>dbRecycleLVs</td>
<td>The RecycleLVs key</td>
</tr>
<tr>
<td>dbFlushTransactionTimeout</td>
<td>The FlushTransactionTimeout key</td>
</tr>
</tbody>
</table>

**Remarks**

Use the `SetOption` method to override registry values at run-time. New values established with the `SetOption` method remain in effect until changed again by another `SetOption` call, or until the `DBEngine` object is closed.

For further details on what the registry keys do, and appropriate values to set them to, see [Initializing the Microsoft Jet 3.5 Database Engine](#) and [Initializing the Microsoft Jet 4.0 Database Engine](#).
## Synchronize Method

Synchronizes two replicas. *(Microsoft Jet databases only).*

### Syntax

```
database.Synchronize pathname, exchange
```

The **Synchronize** method syntax has the following parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>An object variable that represents a Database object that is a replica. A String that contains the path to the target replica with which database will be synchronized. The .mdb file name extension is optional.</td>
</tr>
<tr>
<td>pathname</td>
<td>Optional. A constant indicating which direction to synchronize changes between the two databases, as specified in Settings.</td>
</tr>
</tbody>
</table>

### Settings
You can use the following constants in the exchange argument. You can use one of the first three constants with or without the fourth constant.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbRepExportChanges</code></td>
<td>Sends changes from database to pathname.</td>
</tr>
<tr>
<td><code>dbRepImportChanges</code></td>
<td>Sends changes from pathname to database.</td>
</tr>
<tr>
<td><code>dbRepImpExpChanges</code></td>
<td>(Default) Sends changes from database to pathname, and vice-versa, also known as bidirectional exchange.</td>
</tr>
<tr>
<td><code>dbRepSyncInternet</code></td>
<td>Exchanges data between files connected by an Internet pathway.</td>
</tr>
</tbody>
</table>

**Remarks**

You use **Synchronize** to exchange data and design changes between two databases. Design changes always happen first. Both databases must be at the same design level before they can exchange data. For example, an exchange of type `dbRepExportChanges` might cause design changes at a replica even though data changes flow only from the database to pathname.

The replica identified in `pathname` must be part of the same replica set. If both replicas have the same ReplicaID property setting or are Design Masters for two different replica sets, the synchronization fails.

When you synchronize two replicas over the Internet, you must use the `dbRepSyncInternet` constant. In this case, you specify a Uniform Resource Locator (URL) address for the `pathname` argument instead of specifying a local area network path.

**Note** You can't synchronize partial replicas with other partial replicas. See the PopulatePartial method for more information.

Synchronization over the Internet requires the Replication Manager, which is only available in the Microsoft Office 97, Developer Edition.
Update Method

Saves the contents of the copy buffer to an updatable Recordset object.

Syntax

\texttt{recordset.Update(type, force )}

The \texttt{Update} method syntax has the following parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{recordset}</td>
<td>An object variable that represents an open, updatable Recordset object.</td>
</tr>
<tr>
<td>\texttt{type}</td>
<td>Optional. A constant indicating the type of update, as specified in Settings (ODBCDirect workspaces only).</td>
</tr>
<tr>
<td>\texttt{force}</td>
<td>Optional. A Boolean value indicating whether or not to force the changes into the database, regardless of whether the underlying data has been changed by another user since the AddNew, Delete, or Edit call. If \texttt{True}, the changes are forced and changes made by other users are simply overwritten. If \texttt{False} (default), changes made by another user while the update is pending will cause the update to \textit{aborted}.</td>
</tr>
</tbody>
</table>
fail for those changes that are in conflict. No error occurs, but the BatchCollisionCount and BatchCollisions properties will indicate the number of conflicts and the rows affected by conflicts, respectively (ODBCDirect workspaces only).

Settings

You can use the following values for the type argument. You can use the non-default values only if batch updating is enabled.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbUpdateRegular</td>
<td>Default. Pending changes aren’t cached and are written to disk immediately.</td>
</tr>
<tr>
<td>dbUpdateBatch</td>
<td>All pending changes in the update cache are written to disk.</td>
</tr>
<tr>
<td>dbUpdateCurrentRecord</td>
<td>Only the current record’s pending changes are written to disk.</td>
</tr>
</tbody>
</table>

Remarks

Use Update to save the current record and any changes you've made to it.

Caution  Changes to the current record are lost if:

- You use the Edit or AddNew method, and then move to another record without first using Update.
- You use Edit or AddNew, and then use Edit or AddNew again without first using Update.
- You set the Bookmark property to another record.
- You close the Recordset without first using Update.
- You cancel the Edit operation by using CancelUpdate.

To edit a record, use the Edit method to copy the contents of the current record to the copy buffer. If you don't use Edit first, an error occurs when you use
Update or attempt to change a field's value.

In an ODBCDirect workspace, you can do batch updates, provided the cursor library supports batch updates, and the Recordset was opened with the optimistic batch locking option.

In a Microsoft Jet workspace, when the Recordset object's LockEdits property setting is True (pessimistically locked) in a multiuser environment, the record remains locked from the time Edit is used until the Update method is executed or the edit is canceled. If the LockEdits property setting is False (optimistically locked), the record is locked and compared with the pre-edited record just before it is updated in the database. If the record has changed since you used the Edit method, the Update operation fails. Microsoft Jet-connected ODBC and installable ISAM databases always use optimistic locking. To continue the Update operation with your changes, use the Update method again. To revert to the record as the other user changed it, refresh the current record by using Move 0.

Note  To add, edit, or delete a record, there must be a unique index on the record in the underlying data source. If not, a "Permission denied" error will occur on the AddNew, Delete, or Edit method call in a Microsoft Jet workspace, or an "Invalid argument" error will occur on the Update call in an ODBCDirect workspace.
This reference groups all DAO properties by object or collection. To determine whether a particular property is available to Microsoft Jet or ODBC databases, check the Help topic for that property.

Connection

Parameter

Container

Property
Database  QueryDef

DBEngine  Recordset

Document  Relation

Error  TableDef
<table>
<thead>
<tr>
<th>Field</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>Workspace</td>
</tr>
<tr>
<td>Index</td>
<td></td>
</tr>
</tbody>
</table>
This reference lists alphabetically all DAO properties available to Microsoft Jet workspaces.


A-C

AbsolutePosition  CacheStart
AllowZeroLength  Clustered
AllPermissions  CollatingOrder
Attributes  ConflictTable
BOF  Connect
Bookmark  Container
Bookmarkable  Count
CacheSize


D-H

DataUpdatable  EOF
DateCreated  FieldSize
DefaultUser  Filter
DefaultPassword  Foreign
DefaultValue  ForeignName
Description  ForeignTable
DesignMasterID  HelpContext
DistinctCount  HelpFile
EditMode
I-O

IgnoreNulls
Index
Inherit
Inherited IniPath
IsolateODBCTrans
KeepLocal
LastModified
LastUpdated
LockEdits
LoginTimeout
LogMessages
MaxRecords
Name
NoMatch
Number
ODBCTimeout
OrdinalPosition
Owner
<table>
<thead>
<tr>
<th>PartialReplica</th>
<th>RecordsAffected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password</td>
<td>Replicable</td>
</tr>
<tr>
<td>PercentPosition</td>
<td>ReplicableBool</td>
</tr>
<tr>
<td>Permissions</td>
<td>ReplicaFilter</td>
</tr>
<tr>
<td>PID</td>
<td>ReplicaID</td>
</tr>
<tr>
<td>Primary</td>
<td>Required</td>
</tr>
<tr>
<td>QueryTimeout</td>
<td>Restartable</td>
</tr>
<tr>
<td>RecordCount</td>
<td>ReturnsRecords</td>
</tr>
<tr>
<td><strong>S-Z</strong></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td><strong>Size</strong></td>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><strong>Sort</strong></td>
<td><strong>Unique</strong></td>
</tr>
<tr>
<td><strong>Source</strong></td>
<td><strong>Updatable</strong></td>
</tr>
<tr>
<td><strong>SourceField</strong></td>
<td><strong>UserName</strong></td>
</tr>
<tr>
<td><strong>SourceTable</strong></td>
<td><strong>V1xNullBehavior</strong></td>
</tr>
<tr>
<td><strong>SourceTableName</strong></td>
<td><strong>ValidateOnSet</strong></td>
</tr>
<tr>
<td><strong>SQL</strong></td>
<td><strong>ValidationRule</strong></td>
</tr>
<tr>
<td><strong>SystemDB</strong></td>
<td><strong>ValidationText</strong></td>
</tr>
<tr>
<td><strong>Table</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td><strong>Transactions</strong></td>
<td><strong>Version</strong></td>
</tr>
</tbody>
</table>
DAO Properties for ODBCDirect Workspaces

This reference lists alphabetically all DAO properties available to ODBCDirect workspaces.
<table>
<thead>
<tr>
<th>E-Q</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EditMode</strong></td>
</tr>
<tr>
<td><strong>EOF</strong></td>
</tr>
<tr>
<td><strong>FieldSize</strong></td>
</tr>
<tr>
<td><strong>HelpContext</strong></td>
</tr>
<tr>
<td><strong>HelpFile</strong></td>
</tr>
<tr>
<td><strong>LastModified</strong></td>
</tr>
<tr>
<td><strong>LockEdits</strong></td>
</tr>
<tr>
<td><strong>LoginTimeout</strong></td>
</tr>
<tr>
<td><strong>LogMessages</strong></td>
</tr>
</tbody>
</table>
R-Z

RecordCount    StillExecuting
RecordsAffected Transactions
RecordStatus   Type
Restartable    Updatable
Size           UpdateOptions
Source         UserName
SourceField    Value
SourceTable    Version
SQL            VisibleValue
Absoluteposition Property

Sets or returns the relative record number of a Recordset object's current record.

Settings and Return Values

The setting or return value is a Long integer from 0 to one less than the number of records in the Recordset object. It corresponds to the ordinal position of the current record in the Recordset object specified by the object.

Remarks

You can use the AbsolutePosition property to position the current record pointer to a specific record based on its ordinal position in a dynaset- or snapshot-type Recordset object. You can also determine the current record number by checking the AbsolutePosition property setting.

Because the AbsolutePosition property value is zero-based (that is, a setting of 0 refers to the first record in the Recordset object), you cannot set it to a value greater than or equal to the number of populated records; doing so causes a trappable error. You can determine the number of populated records in the Recordset object by checking the RecordCount property setting. The maximum allowable setting for the AbsolutePosition property is the value of the
**RecordCount** property minus 1.

If there is no current record, as when there are no records in the **Recordset** object, **AbsolutePosition** returns –1. If the current record is deleted, the **AbsolutePosition** property value isn't defined, and a trappable error occurs if it's referenced. New records are added to the end of the sequence.

You shouldn't use this property as a surrogate record number. **Bookmarks** are still the recommended way of retaining and returning to a given position and are the only way to position the current record across all types of **Recordset** objects. In particular, the position of a record changes when one or more records preceding it are deleted. There is also no assurance that a record will have the same absolute position if the **Recordset** object is re-created again because the order of individual records within a **Recordset** object isn't guaranteed unless it's created with an **SQL statement** by using an ORDER BY clause.

**Notes**

- Setting the **AbsolutePosition** property to a value greater than zero on a newly opened but unpopulated **Recordset** object causes a trappable error. Populate the **Recordset** object first with the **MoveLast** method.

- The **AbsolutePosition** property isn't available on **forward-only–type Recordset** objects, or on **Recordset** objects opened from **pass-through queries** against Microsoft Jet-connected ODBC databases.
AllowZeroLength Property

Sets or returns a value that indicates whether a zero-length string (""") is a valid setting for the Value property of the Field object with a Text or Memo data type (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a Boolean data type that indicates if a value is valid. The value is True if the Field object accepts a zero-length string as its Value property; the default value is False.

Remarks

For an object not yet appended to the Fields collection, this property is read/write.

Once appended to a Fields collection, the availability of the AllowZeroLength property depends on the object that contains the Fields collection, as shown in the following table.

<table>
<thead>
<tr>
<th>If the Fields collection belongs to an</th>
<th>Then AllowZeroLength is</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Object</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index object</td>
<td>Not supported</td>
</tr>
<tr>
<td>QueryDef object</td>
<td>Read-only</td>
</tr>
<tr>
<td>Recordset object</td>
<td>Read-only</td>
</tr>
<tr>
<td>Relation object</td>
<td>Not supported</td>
</tr>
<tr>
<td>TableDef object</td>
<td>Read/write</td>
</tr>
</tbody>
</table>

You can use this property along with the [Required](#), [ValidateOnSet](#), or [ValidationRule](#) property to validate a value in a field.
AllPermissions Property

Returns all the permissions that apply to the current **UserName** property of the **Container** or **Document** object, including **permissions** that are specific to the user as well as the permissions a user inherits from memberships in **groups** (**Microsoft Jet workspaces** only).

**Return Values**

For any **Container** or **Document** object, the return value is a **Long** value or constant(s) that may include the following.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecReadDef</td>
<td>The user can read the table definition, including column and index information.</td>
</tr>
<tr>
<td>dbSecWriteDef</td>
<td>The user can modify or delete the table definition, including column and index information.</td>
</tr>
<tr>
<td>dbSecRetrieveData</td>
<td>The user can retrieve data from the <strong>Document</strong> object.</td>
</tr>
<tr>
<td>dbSecInsertData</td>
<td>The user can add records.</td>
</tr>
</tbody>
</table>
**dbSecReplaceData**  The user can modify records.

**dbSecDeleteData**  The user can delete records.

In addition, the Databases container or any **Document** object in a **Documents** collection may include the following.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecDeleteData</td>
<td>The user can delete records.</td>
</tr>
<tr>
<td>dbSecDBAdmin</td>
<td>The user can <a href="#">replicate</a> the database and change the database password.</td>
</tr>
<tr>
<td></td>
<td>The user can create new databases. This setting is valid only on the Databases container in the workgroup information file (System.mdw).</td>
</tr>
<tr>
<td>dbSecDBExclusive</td>
<td>The user has <a href="#">exclusive</a> access to the database.</td>
</tr>
<tr>
<td>dbSecDBOpen</td>
<td>The user can open the database.</td>
</tr>
</tbody>
</table>

**Remarks**

This property contrasts with the **Permissions** property, which returns only the permissions that are specific to the user and doesn't include any permissions that the user may also have as a member of groups. If the current value of the **UserName** property is a group, then the **AllPermissions** property returns the same values as the **Permissions** property.
Attributes Property

Sets or returns a value that indicates one or more characteristics of a Field, Relation, or TableDef object.

Settings and Return Values

The setting or return value is Long data type, and the default value is 0.

For a Field object, the value specifies characteristics of the field represented by the Field object and can be a combination of these constants.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbAutoIncrField</td>
<td>The field value for new records is automatically incremented to a unique Long integer that can't be changed (in a Microsoft Jet workspace, supported only for Microsoft Jet database(.mdb) tables). The field is sorted in descending (Z to A or 100 to 0) order; this option applies only to a Field object in a Fields collection of an Index object. If you</td>
</tr>
</tbody>
</table>
**dbDescending**

omit this constant, the field is sorted in ascending (A to Z or 0 to 100) order. This is the default value for **Index** and **TableDef** fields (Microsoft Jet workspaces only).

**dbFixedField**

The field size is fixed (default for Numeric fields).

**dbHyperlinkField**

The field contains hyperlink information (Memo fields only).

**dbSystemField**

The field stores replication information for **replicas**; you can't delete this type of field (Microsoft Jet workspaces only).

**dbUpdatableField**

The field value can be changed.

**dbVariableField**

The field size is variable (Text fields only).

For a **Relation** object, the value specifies characteristics of the relationship represented by the **Relation** object and can be a combination of these constants.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbRelationUnique</td>
<td>The relationship is <strong>one-to-one</strong>.</td>
</tr>
<tr>
<td>dbRelationDontEnforce</td>
<td>The relationship isn't enforced (no referential integrity).</td>
</tr>
<tr>
<td>dbRelationInherited</td>
<td>The relationship exists in a non-<strong>current database</strong> that contains the two linked tables.</td>
</tr>
<tr>
<td>dbRelationUpdateCascade</td>
<td>Updates will cascade.</td>
</tr>
<tr>
<td>dbRelationDeleteCascade</td>
<td>Deletions will cascade.</td>
</tr>
<tr>
<td>dbRelationLeft</td>
<td>Microsoft Access only. In Design view, display a LEFT JOIN as the default join type.</td>
</tr>
<tr>
<td>dbRelationRight</td>
<td>Microsoft Access only. In Design view, display a RIGHT JOIN as the default join type.</td>
</tr>
</tbody>
</table>
**Note** If you set the **Relation** object's **Attributes** property to activate cascading operations, the **Microsoft Jet database engine** automatically updates or deletes records in one or more other tables when changes occur in related **primary tables**.

For example, suppose you establish a **cascading delete** relationship between a Customers table and an Orders table. When you delete records from the Customers table, records in the Orders table related to that customer are also deleted. In addition, if you establish cascading delete relationships between the Orders table and other tables, records from those tables are automatically deleted when you delete records from the Customers table.

For a **TableDef** object, the value specifies characteristics of the table represented by the **TableDef** object and can be a combination of these **Long** constants.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbAttachExclusive</td>
<td>For databases that use the Microsoft Jet database engine, the table is a <strong>linked table</strong> opened for exclusive use. You can set this constant on an appended <strong>TableDef</strong> object for a local table, but not on a remote table.</td>
</tr>
<tr>
<td>dbAttachSavePWD</td>
<td>For databases that use the Microsoft Jet database engine, the user ID and password for the remotely linked table are saved with the connection information. You can set this constant on an appended <strong>TableDef</strong> object for a remote table, but not on a local table.</td>
</tr>
<tr>
<td>dbSystemObject</td>
<td>The table is a system table provided by the Microsoft Jet database engine. You can set this constant on an appended <strong>TableDef</strong> object.</td>
</tr>
<tr>
<td>dbHiddenObject</td>
<td>The table is a hidden table provided by the Microsoft Jet database engine. You can set this constant on an appended <strong>TableDef</strong> object.</td>
</tr>
</tbody>
</table>

The table is a linked table from a non-
<table>
<thead>
<tr>
<th>dbAttachedTable</th>
<th>ODBC data source such as a Microsoft Jet or Paradox database (read-only). The table is a linked table from an ODBC data source, such as Microsoft SQL Server (read-only).</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbAttachedODBC</td>
<td></td>
</tr>
</tbody>
</table>

**Remarks**

For an object not yet appended to a collection, this property is read/write.

For an appended **Field** object, the availability of the **Attributes** property depends on the object that contains the **Fields** collection.

<table>
<thead>
<tr>
<th>If the Field object belongs to an</th>
<th>Then Attributes is</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Index</strong> object</td>
<td>Read/write until the <strong>TableDef</strong> object that the <strong>Index</strong> object is appended to is appended to a <strong>Database</strong> object; then the property is read-only.</td>
</tr>
<tr>
<td><strong>QueryDef</strong> object</td>
<td>Read-only</td>
</tr>
<tr>
<td><strong>Recordset</strong> object</td>
<td>Read-only</td>
</tr>
<tr>
<td><strong>Relation</strong> object</td>
<td>Not supported</td>
</tr>
<tr>
<td><strong>TableDef</strong> object</td>
<td>Read/write</td>
</tr>
</tbody>
</table>

For an appended **Relation** object, the **Attributes** property setting is read-only.

For an appended **TableDef** object, the property is read/write, although you can't set all of the constants if the object is appended, as noted in Settings and Return Values.

When you set multiple attributes, you can combine them by summing the appropriate constants. Any invalid values are ignored without producing an error.
BatchCollisionCount Property

Returns the number of records that did not complete the last batch update (ODBCDirect workspaces only).

**Return Value**

The return value is a Long that indicates the number of failing records, or 0 if all records were successfully updated.

**Remarks**

This property indicates how many records encountered collisions or otherwise failed to update during the last batch update attempt. The value of this property corresponds to the number of bookmarks in the BatchCollisions property.

If you set the working Recordset object's Bookmark property to bookmark values in the BatchCollisions array, you can move to each record that failed to complete the most recent batch Update operation.

After the collision records are corrected, a batch-mode Update method can be called again. At this point DAO attempts another batch update, and the BatchCollisions property again reflects the set of records that failed the second
attempt. Any records that succeeded in the previous attempt are not sent in the current attempt, because they now have a RecordStatus property of dbRecordUnmodified. This process can continue as long as collisions occur, or until you abandon the updates and close the result set.
BatchCollisions Property

Returns an array of bookmarks indicating the rows that generated collisions in the last batch update operation (ODBCDirect workspaces only).

Return Value

The return value is a variant expression containing an array of bookmarks.

Remarks

This property contains an array of bookmarks to rows that encountered a collision during the last attempted batch Update call. The BatchCollisionCount property indicates the number of elements in the array.

If you set the working Recordset object's Bookmark property to bookmark values in the BatchCollisions array, you can move to each record that failed to complete the most recent batch-mode Update operation.

After the collision records are corrected, you can call the batch mode Update method again. At this point DAO attempts another batch update, and the BatchCollisions property again reflects the set of records that failed the second attempt. Any records that succeeded in the previous attempt are not sent in the
current attempt, as they now have a RecordStatus property of dbRecordUnmodified. This process can continue as long as collisions occur, or until you abandon the updates and close the result set.

This array is re-created each time you execute a batch-mode Update method.
BatchSize Property

Sets or returns the number of statements sent back to the server in each batch (ODBCDirect workspaces only).

Settings And Return Values

The setting or return value is a Long that indicates the number of batched statements sent the server in a single batch update. The default value is 15.

Remarks

The BatchSize property determines the batch size used when sending statements to the server in a batch update. The value of the property determines the number of statements sent to the server in one command buffer. By default, 15 statements are sent to the server in each batch. This property can be changed at any time. If a database server doesn't support statement batching, you can set this property to 1, causing each statement to be sent separately.
BOF, EOF Properties

- **BOF** returns a value that indicates whether the current record position is before the first record in a **Recordset** object.

- **EOF** returns a value that indicates whether the current record position is after the last record in a **Recordset** object.

**Return Values**

The return values for the **BOF** and **EOF** properties are **Boolean** values.

The **BOF** property returns **True** if the current record position is before the first record, and **False** if the current record position is on or after the first record.

The **EOF** property returns **True** if the current record position is after the last record, and **False** if the current record position is on or before the last record.

**Remarks**

You can use the **BOF** and **EOF** properties to determine whether a **Recordset** object contains records or whether you've gone beyond the limits of a **Recordset** object when you move from record to record.
The location of the current record pointer determines the **BOF** and **EOF** return values.

If either the **BOF** or **EOF** property is **True**, there is no current record.

If you open a **Recordset** object containing no records, the **BOF** and **EOF** properties are set to **True**, and the **Recordset** object's **RecordCount** property setting is 0. When you open a **Recordset** object that contains at least one record, the first record is the current record and the **BOF** and **EOF** properties are **False**; they remain **False** until you move beyond the beginning or end of the **Recordset** object by using the **MovePrevious** or **MoveNext** method, respectively. When you move beyond the beginning or end of the **Recordset**, there is no current record or no record exists.

If you delete the last remaining record in the **Recordset** object, the **BOF** and **EOF** properties may remain **False** until you attempt to reposition the current record.

If you use the **MoveLast** method on a **Recordset** object containing records, the last record becomes the current record; if you then use the **MoveNext** method, the current record becomes invalid and the **EOF** property is set to **True**. Conversely, if you use the **MoveFirst** method on a **Recordset** object containing records, the first record becomes the current record; if you then use the **MovePrevious** method, there is no current record and the **BOF** property is set to **True**.

Typically, when you work with all the records in a **Recordset** object, your code will loop through the records by using the **MoveNext** method until the **EOF** property is set to **True**.

If you use the **MoveNext** method while the **EOF** property is set to **True** or the **MovePrevious** method while the **BOF** property is set to **True**, an error occurs.

This table shows which Move methods are allowed with different combinations of the **BOF** and **EOF** properties.

<table>
<thead>
<tr>
<th>MoveFirst, MoveLast</th>
<th>MovePrevious, Move &lt; 0</th>
<th>Move 0</th>
<th>MoveNext, Move &gt; 0</th>
</tr>
</thead>
</table>
| BOF=True,

<table>
<thead>
<tr>
<th>MovePrevious, Move &lt; 0</th>
<th>Move 0</th>
<th>MoveNext, Move &gt; 0</th>
</tr>
</thead>
</table>
| BOF=True,

This table shows which Move methods are allowed with different combinations of the **BOF** and **EOF** properties.
Allowing a Move method doesn't mean that the method will successfully locate a record. It merely indicates that an attempt to perform the specified Move method is allowed and won't generate an error. The state of the BOF and EOF properties may change as a result of the attempted Move.

An OpenRecordset method internally invokes a MoveFirst method. Therefore, using an OpenRecordset method on an empty set of records sets the BOF and EOF properties to True. (See the following table for the behavior of a failed MoveFirst method.)

All Move methods that successfully locate a record will set both BOF and EOF to False.

In a Microsoft Jet workspace, if you add a record to an empty Recordset, BOF will become False, but EOF will remain True, indicating that the current position is at the end of Recordset. In an ODBCDirect workspace, both BOF and EOF will become False, indicating that the current position is on the new record.

Any Delete method, even if it removes the only remaining record from a Recordset, won't change the setting of the BOF or EOF property.

The following table shows how Move methods that don't locate a record affect the BOF and EOF property settings.

<table>
<thead>
<tr>
<th>Move Method</th>
<th>BOF</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoveFirst, MoveLast</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Move 0</td>
<td>No change</td>
<td>No change</td>
</tr>
<tr>
<td>MovePrevious, Move &lt; 0</td>
<td>True</td>
<td>No change</td>
</tr>
<tr>
<td>MoveNext, Move &gt; 0</td>
<td>No change</td>
<td>True</td>
</tr>
</tbody>
</table>
Bookmark Property

Sets or returns a bookmark that uniquely identifies the current record in a Recordset object.

Settings and Return Values

The setting or return value is a string expression or variant expression that evaluates to a valid bookmark. The data type is a Variant array of Byte data.

Remarks

For a Recordset object based entirely on Microsoft Jet tables, the value of the Bookmarkable property is True, and you can use the Bookmark property with that Recordset. Other database products may not support bookmarks, however. For example, you can't use bookmarks in any Recordset object based on a linked Paradox table that has no primary key.

When you create or open a Recordset object, each of its records already has a unique bookmark. You can save the bookmark for the current record by assigning the value of the Bookmark property to a variable. To quickly return to that record at any time after moving to a different record, set the Recordset object's Bookmark property to the value of that variable.
There is no limit to the number of bookmarks you can establish. To create a bookmark for a record other than the current record, move to the desired record and assign the value of the **Bookmark** property to a **String** variable that identifies the record.

To make sure the **Recordset** object supports bookmarks, check the value of its **Bookmarkable** property before you use the **Bookmark** property. If the **Bookmarkable** property is **False**, the **Recordset** object doesn't support bookmarks, and using the **Bookmark** property results in a trappable error.

If you use the **Clone** method to create a copy of a **Recordset** object, the **Bookmark** property settings for the original and the duplicate **Recordset** objects are identical and can be used interchangeably. However, you can't use bookmarks from different **Recordset** objects interchangeably, even if they were created by using the same object or the same **SQL statement**.

If you set the **Bookmark** property to a value that represents a deleted record, a trappable error occurs.

The value of the **Bookmark** property isn't the same as a record number.
Returns a value that indicates whether a **Recordset** object supports **bookmarks**, which you can set by using the **Bookmark** property.

**Return Values**

The return value is a **Boolean** data type that returns **True** if the object supports bookmarks.

**Remarks**

Check the **bookmarkable** property setting of a **Recordset** object before you attempt to set or check the **Bookmark** property.

For **Recordset** objects based entirely on **Microsoft Jet** tables, the value of the **Bookmarkable** property is **True**, and you can use bookmarks. Other database products may not support bookmarks, however. For example, you can't use bookmarks in any **Recordset** object based on a **linked** Paradox table that has no **primary key**.
CacheSize Property

Sets or returns the number of records retrieved from an ODBC data source that will be cached locally.

Settings and Return Values

The setting or return value is a Long value and must be between 5 and 1200, but not greater than available memory will allow. A typical value is 100. A setting of 0 turns off caching.

Remarks

Data caching improves performance if you use Recordset objects to retrieve data from a remote server. A cache is a space in local memory that holds the data most recently retrieved from the server; this is useful if users request the data again while the application is running. When users request data, the Microsoft Jet database engine checks the cache for the requested data first rather than retrieving it from the server, which takes more time. The cache only saves data that comes from an ODBC data source.

Any Microsoft Jet-connected ODBC data source, such as a linked table, can have a local cache. To create the cache, open a Recordset object from the
remote data source, set the `CacheSize` and `CacheStart` properties, and then use the `FillCache` method, or step through the records by using the Move methods.

An `ODBCDirect` workspace can use a local cache. To create the cache, set the `CacheSize` property on a `QueryDef` object. On a `Relation` object, `CacheSize` is read-only and depends on the value of the `QueryDef` object's `CacheSize` property. You can't use the `CacheStart` property on `FillCache` method in an `ODBCDirect` workspace. In a Microsoft Jet workspace, the `CacheSize` property is not available on a `QueryDef` object.

You can base the `CacheSize` property setting on the number of records your application can handle at one time. For example, if you're using a `Recordset` object as the source of the data to be displayed on screen, you could set its `CacheSize` property to 20 to display 20 records at one time.

The Microsoft Jet database engine requests records within the cache range from the cache, and it requests records outside the cache range from the server.

Records retrieved from the cache don't reflect concurrent changes that other users made to the source data.

To force an update of all the cached data, set the `CacheSize` property of the `Recordset` object to 0, re-set it to the size of the cache you originally requested, and then use the `FillCache` method.
CacheStart Property

Sets or returns a value that specifies the `bookmark` of the first record in a `dynaset-type Recordset` object containing data to be locally cached from an `ODBC data source` (`Microsoft Jet workspaces` only).

**Settings And Return Values**

The setting or return value is a `String` that specifies a bookmark.

**Remarks**

Data caching improves the performance of an application that retrieves data from a remote server through dynaset-type `Recordset` objects. A cache is a space in local memory that holds the data most recently retrieved from the server in the event that the data will be requested again while the application is running. When data is requested, the `Microsoft Jet database engine` checks the cache for the requested data first rather than retrieving it from the server, which takes more time. Only data from an ODBC data source can be saved in the cache.

Any Microsoft Jet-connected ODBC data source, such as a `linked table`, can have a local cache. To create the cache, open a `Recordset` object from the remote data source, set the `CacheSize` and `CacheStart` properties, and then use
the FillCache method or step through the records using the Move methods.

The CacheStart property setting is the bookmark of the first record in the Recordset object to be cached. You can use the bookmark of any record to set the CacheStart property. Make the record you want to start the cache the current record, and set the CacheStart property equal to the Bookmark property.

The Microsoft Jet database engine requests records within the cache range from the cache, and it requests records outside the cache range from the server.

Records retrieved from the cache don't reflect changes made concurrently to the source data by other users.

To force an update of all the cached data, set the CacheSize property of the Recordset object to 0, set it to the size of the cache you originally requested, and then use the FillCache method.
Clustered Property

Sets or returns a value that indicates whether an Index object represents a clustered index for a table (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a Boolean data type that is True if the Index object represents a clustered index.

Remarks

Some ISAM desktop database formats use clustered indexes. A clustered index consists of one or more nonkey fields that, taken together, arrange all records in a table in a predefined order. A clustered index provides efficient access to records in a table in which the index values may not be unique.

The Clustered property is read/write for a new Index object not yet appended to a collection and read-only for an existing Index object in an Indexes collection.

Notes

- Microsoft Jet databases ignore the Clustered property because the
Microsoft Jet database engine doesn't support clustered indexes.

- For [ODBC data sources](#) the **Clustered** property always returns **False**; it does not detect whether or not the ODBC data source has a clustered index.
CollatingOrder Property

Returns a value that specifies the sequence of the sort order in text for string comparison or sorting (Microsoft Jet workspaces only).

Return Values

The return value is a Long value or constant that can be one of the following values.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Sort order</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSortGeneral</td>
<td>General (English, French, German, Portuguese, Italian, and Modern Spanish)</td>
</tr>
<tr>
<td>dbSortArabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>dbSortChineseSimplified</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>dbSortChineseTraditional</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>dbSortCyrillic</td>
<td>Russian</td>
</tr>
<tr>
<td>dbSortCzech</td>
<td>Czech</td>
</tr>
<tr>
<td>dbSortDutch</td>
<td>Dutch</td>
</tr>
<tr>
<td>dbSortGreek</td>
<td>Greek</td>
</tr>
</tbody>
</table>
dbSortHebrew Hebrew
dbSortHungarian Hungarian
dbSortIcelandic Icelandic
dbSortJapanese Japanese
dbSortKorean Korean
dbSortNeutral Neutral
dbSortNorwDan Norwegian or Danish
dbSortPDXIntl Paradox International
dbSortPDXNor Paradox Norwegian or Danish
dbSortPDXSwe Paradox Swedish or Finnish
dbSortPolish Polish
dbSortSlovenian Slovenian
dbSortSpanish Spanish
dbSortSwedFin Swedish or Finnish
dbSortThai Thai
dbSortTurkish Turkish
dbSortUndefined Undefined or unknown

Remarks

The availability of the **CollatingOrder** property depends on the object that contains the **Fields** collection, as shown in the following table.

<table>
<thead>
<tr>
<th>If the Fields collection belongs to an object</th>
<th>Then CollatingOrder is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index object</td>
<td>Not supported</td>
</tr>
<tr>
<td>QueryDef object</td>
<td>Read-only</td>
</tr>
<tr>
<td>Recordset object</td>
<td>Read-only</td>
</tr>
<tr>
<td>Relation object</td>
<td>Not supported</td>
</tr>
<tr>
<td>TableDef object</td>
<td>Read-only</td>
</tr>
</tbody>
</table>

The **CollatingOrder** property setting corresponds to the `locale` argument of the **CreateDatabase** method when the database was created or the **CompactDatabase** method when the database was most recently compacted.
Check the **CollatingOrder** property setting of a **Database** or **Field** object to determine the string comparison method for the database or field. You can set the **CollatingOrder** property of a new, unappended **Field** object if you want the setting of the **Field** object to differ from that of the **Database** object that contains it.

The **CollatingOrder** and **Attributes** property settings of a **Field** object in a **Fields** collection of an **Index** object together determine the sequence and direction of the sort order in an index. However, you can't set a collating order for an individual index—you can only set it for an entire table.
ConflictTable Property

Returns the name of a conflict table containing the database records that conflicted during the synchronization of two replicas (Microsoft Jet workspaces only).

Return Values

The return value is a String data type that is a zero-length string if there is no conflict table or the database isn't a replica.

Remarks

If two users at two separate replicas each make a change to the same record in the database, the changes made by one user will fail to be applied to the other replica. Consequently, the user with the failed change must resolve the conflicts.

Conflicts occur at the record level, not between fields. For example, if one user changes the Address field and another updates the Phone field in the same record, then one change is rejected. Because conflicts occur at the record level, the rejection occurs even though the successful change and the rejected change are unlikely to result in a true conflict of information.
The synchronization mechanism handles the record conflicts by creating conflict tables, which contain the information that would have been placed in the table, if the change had been successful. You can examine these conflict tables and work through them row by row, fixing whatever is appropriate.

All conflict tables are named `table_conflict`, where `table` is the original name of the table, truncated to the maximum table name length.
Connect Property

Sets or returns a value that provides information about the source of an open connection, an open database, a database used in a pass-through query, or a linked table. For Database objects, new Connection objects, linked tables, and TableDef objects not yet appended to a collection, this property setting is read/write. For QueryDef objects and base tables, this property is read-only.

Syntax

```
object.Connect = databasetype;parameters;
```

The Connect property syntax has these parts.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>An object expression that evaluates to an object in the Applies To list.</td>
</tr>
<tr>
<td><code>databasetype</code></td>
<td>Optional. A String that specifies a database type. For Microsoft Jet databases, exclude this argument; if you specify <code>parameters</code>, use a semicolon (;) as a placeholder.</td>
</tr>
</tbody>
</table>

Optional. A String that specifies additional
parameters	parameters to pass to ODBC or installable ISAM drivers. Use semicolons to separate parameters.

Settings

The **Connect** property setting is a **String** composed of a database type specifier and zero or more parameters separated by semicolons. The **Connect** property passes additional information to ODBC and certain ISAM drivers as needed.

To perform an SQL pass-through query on a table linked to your Microsoft Jet database (.mdb) file, you must first set the **Connect** property of the linked table's database to a valid ODBC connection string.

For a **TableDef** object that represents a linked table, the **Connect** property setting consists of one or two parts (a database type specifier and a path to the database), each of which ends with a semicolon.

The path as shown in the following table is the full path for the directory containing the database files and must be preceded by the identifier DATABASE=. In some cases (as with Microsoft Excel and Microsoft Jet databases), you should include a specific file name in the database path argument.

The following table shows possible database types and their corresponding database specifiers and paths for the **Connect** property setting. In an ODBC Direct workspace, only the "ODBC" specifier can be used.

<table>
<thead>
<tr>
<th>Database type</th>
<th>Specifier</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Jet Database</td>
<td>[database];</td>
<td>drive:\path\filename.mdb</td>
</tr>
<tr>
<td>dBASE III</td>
<td>dBASE III;</td>
<td>drive:\path</td>
</tr>
<tr>
<td>dBASE IV</td>
<td>dBASE IV;</td>
<td>drive:\path</td>
</tr>
<tr>
<td>dBASE 5</td>
<td>dBASE 5.0;</td>
<td>drive:\path</td>
</tr>
<tr>
<td>Paradox 3.x</td>
<td>Paradox 3.x;</td>
<td>drive:\path</td>
</tr>
<tr>
<td>Paradox 4.x</td>
<td>Paradox 4.x;</td>
<td>drive:\path</td>
</tr>
<tr>
<td>Paradox 5.x</td>
<td>Paradox 5.x;</td>
<td>drive:\path</td>
</tr>
<tr>
<td>Microsoft Excel 3.0</td>
<td>Excel 3.0;</td>
<td>drive:\path\filename.xls</td>
</tr>
<tr>
<td>Microsoft Excel 4.0</td>
<td>Excel 4.0;</td>
<td>drive:\path\filename.xls</td>
</tr>
</tbody>
</table>
### Microsoft Excel

- **Excel 5.0;**  
  *drive:\path\filename.xls*

- **Excel 8.0;**  
  *drive:\path\filename.xls*

### Lotus 1-2-3 WKS and WK1

- **Lotus WK1;**  
  *drive:\path\filename.wk1*

- **Lotus WK3;**  
  *drive:\path\filename.wk3*

- **Lotus WK4;**  
  *drive:\path\filename.wk4*

### HTML Import and Export

- **HTML Import;**  
  *drive:\path\filename*

- **HTML Export;**  
  *drive:\path\filename*

### Text

- **Text;**  
  *drive:\path\filename*

### ODBC

- **ODBC;**  
  DATABASE=*database*;  
  UID=*user*;  
  PWD=*password*;  
  DSN=*datasourcename*;  
  [LOGINTIMEOUT=seconds;]

- **None**

### Microsoft Exchange

- **[TABLETYPE={0 | 1}];**  
  *drive:\path\filename.mdb*

- **[PROFILE=profile;]**  
  [PWD=*password*;]  
  [DATABASE=*database*;]

### Remarks

If the specifier is only "ODBC;", the **ODBC driver** displays a dialog box listing all registered **ODBC data source** names so that the user can select a database.

Microsoft Jet 4.0 no longer supports the FoxPro ISAM. Use the FoxPro ODBC driver instead.

If a password is required but not provided in the **Connect** property setting, a login dialog box is displayed the first time a table is accessed by the ODBC
driver and again if the connection is closed and reopened.

For data in Microsoft Exchange, the required MAPILEVEL key should be set to a fully-resolved folder path (for example, "Mailbox - Pat Smith\Alpha/Today"). The path does not include the name of the folder that will be opened as a table; that folder’s name should instead be specified as the name argument to the CreateTable method. The TABLETYPE key should be set to "0" to open a folder (default) or "1" to open an address book. The PROFILE key defaults to the profile currently in use.

For base tables in a Microsoft Jet database (.mdb), the Connect property setting is a zero-length string ("").

You can set the Connect property for a Database object by providing a source argument to the OpenDatabase method. You can check the setting to determine the type, path, user ID, password, or ODBC data source of the database.

On a QueryDef object in a Microsoft Jet workspace, you can use the Connect property with the ReturnsRecords property to create an ODBC SQL pass-through query. The databasetype of the connection string is "ODBC; ", and the remainder of the string contains information specific to the ODBC driver used to access the remote data. For more information, see the documentation for the specific driver.

Notes

- You must set the Connect property before you set the ReturnsRecords property.

- You must have access permissions to the computer that contains the database server you're trying to access.
Connection Property

On a **Database** object, returns the **Connection** object that corresponds to the database (**ODBCDirect workspaces** only).

On a **Recordset** object, returns the **Connection** object that owns the **Recordset** (ODBCDirect workspaces only).

**Settings And Return Values**

The return value is an **object variable** that represents the **Connection**. On a **Database** object, the **Connection** property is read-only, while on a **Recordset** object the property is read-write.

**Remarks**

On a **Database** object, use the **Connection** property to obtain a reference to a **Connection** object that corresponds to the **Database**. In DAO, a **Connection** object and its corresponding **Database** object are simply two different object variable references to the same object. The **Database** property of a **Connection** object and the **Connection** property of a **Database** object make it easier to change connections to an **ODBC data source** through the **Microsoft Jet database engine** to use **ODBCDirect**.
Container Property

Returns the name of the Container object to which a Document object belongs (Microsoft Jet workspaces only).

Return Values

The return value is a String data type.
Count Property

Returns the number of objects in a collection.

Return Value

The return value is an Integer data type.

Remarks

Because members of a collection begin with 0, you should always code loops starting with the 0 member and ending with the value of the Count property minus 1. If you want to loop through the members of a collection without checking the Count property, you can use a For Each...Next command.

The Count property setting is never Null. If its value is 0, there are no objects in the collection.
Returns the Database object that corresponds to this connection (ODBCDirect workspaces only).

**Return Values**

The return value is an object variable that represents a Database object.

**Remarks**

On a Connection object, use the Database property to obtain a reference to a Database object that corresponds to the Connection. In DAO, a Connection object and its corresponding Database object are simply two different object variable references to the same object. The Database property of a Connection object and the Connection property of a Database object make it easier to change connections to an ODBC data source through the Microsoft Jet database engine to use ODBCDirect.
DataUpdatable Property

Returns a value that indicates whether the data in the field represented by a Field object is updatable.

Return Values

The return value is a Boolean data type that returns True if the data in the field is updatable.

Remarks

Use this property to determine whether you can change the Value property setting of a Field object. This property is always False on a Field object whose Attributes property is dbAutoIncrField.

You can use the DataUpdatable property on Field objects that are appended to the Fields collection of QueryDef, Recordset, and Relation objects, but not the Fields collection of Index or TableDef objects.
DateCreated, LastUpdated Properties

- **DateCreated** returns the date and time that an object was created, or the date and time a base table was created if the object is a table-type Recordset object (Microsoft Jet workspaces only).

- **LastUpdated** returns the date and time of the most recent change made to an object, or to a base table if the object is a table-type Recordset object (Microsoft Jet workspaces only).

**Return Values**

The return value is a **Variant** (Date/Time subtype).

**Remarks**

For table-type Recordset objects, the date and time settings are derived from the computer on which the base table was created or last updated. For other objects, **DateCreated** and **LastUpdated** return the date and time that the object was created or last updated. In a multiuser environment, users should get these settings directly from the file server to avoid discrepancies in the **DateCreated** and **LastUpdated** property settings.
DefaultCursorDriver Property

Sets or returns the type of cursor driver used on the connection created by the OpenConnection or OpenDatabase methods (ODBCDirect workspaces only).

Settings And Return Values

The setting or return value is a Long that can be set to one of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbUseDefaultCursor</td>
<td>(Default) Uses server-side cursors if the server supports them; otherwise use the ODBC Cursor Library.</td>
</tr>
<tr>
<td>dbUseODBCCursor</td>
<td>Always uses the ODBC Cursor Library. This option provides better performance for small result sets, but degrades quickly for larger result sets.</td>
</tr>
<tr>
<td>dbUseServerCursor</td>
<td>Always uses server-side cursors. For most large operations this option provides better performance, but might cause more network traffic.</td>
</tr>
</tbody>
</table>
**dbUseClientBatchCursor**

Always uses the client batch cursor library. This option is required for [batch updates](#). Opens all cursors (that is, Recordset objects) as forward-only type, read-only, with a rowset size of 1. Also known as "cursorless queries."

**dbUseNoCursor**

Remarks

This property setting only affects connections established after the property has been set. Changing the `DefaultCursorDriver` property has no effect on existing connections.
DefaultType Property

Sets or returns a value that indicates what type of workspace (Microsoft Jet or ODBCDirect) will be used by the next Workspace object created.

Settings And Return Values

The setting or return value is a Long that can be set to either of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbUseJet</td>
<td>Creates Workspace objects connected to the Microsoft Jet database engine</td>
</tr>
<tr>
<td>dbUseODBC</td>
<td>Creates Workspace objects connected to an ODBC data source</td>
</tr>
</tbody>
</table>

Remarks

The setting can be overridden for a single Workspace by setting the type argument to the CreateWorkspace method.
DefaultUser, DefaultPassword Properties

- **DefaultUser** sets the user name used to create the default Workspace when it is initialized.

- **DefaultPassword** sets the password used to create the default Workspace when it is initialized.

**Settings**

The setting for **DefaultUser** is a [String](#) data type. It can be 1–20 characters long in [Microsoft Jet workspaces](#) and any length in [ODBCDirect workspaces](#), and it can include alphabetic characters, accented characters, numbers, spaces, and symbols except for: " (quotation marks), / (forward slash), \ (backslash), [ ] (brackets), : (colon), | (pipe), < (less-than sign), > (greater-than sign), + (plus sign), = (equal sign), ; (semicolon), , (comma), ? (question mark), * (asterisk), leading spaces, and control characters (ASCII 00 to ASCII 31).

The setting for **DefaultPassword** is a [String](#) data type that can be up to 14 characters long in Microsoft Jet databases and any length in ODBCDirect
connections. It can contain any character except ASCII 0.

By default, the **DefaultUser** property is set to "admin" and the **DefaultPassword** property is set to a zero-length string ("").

**Remarks**

User names aren't usually case-sensitive; however, if you're re-creating a user account that was deleted or created in a different workgroup, the user name must be an exact case-sensitive match of the original name. Passwords are case-sensitive.

Typically, you use the **CreateWorkspace** method to create a **Workspace** object with a given user name and password. However, for backward compatibility with earlier versions and for convenience when you don't implement a secured database, the **Microsoft Jet database engine** automatically creates a default **Workspace** object when needed if one isn't already open. In this case, the **DefaultUser** and **DefaultPassword** property values define the user and password for the default **Workspace** object.

For this property to take effect, you should set it before calling any DAO methods.
DefaultValue Property

Sets or returns the default value of a Field object. For a Field object not yet appended to the Fields collection, this property is read/write (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String data type that can contain a maximum of 255 characters. It can be either text or an expression. If the property setting is an expression, it can't contain user-defined functions, Microsoft Jet database engine SQL aggregate functions, or references to queries, forms, or other Field objects.

Note  You can also set the DefaultValue property of a Field object on a TableDef object to a special value called "GenUniqueID()". This causes a random number to be assigned to this field whenever a new record is added or created, thereby giving each record a unique identifier. The field's Type property must be Long.

Remarks

The availability of the DefaultValue property depends on the object that contains the Fields collection, as shown in the following table.
If the Fields collection belongs to an Index object, the DefaultValue is Not supported. If it belongs to a QueryDef object, it is Read-only. If it belongs to a Recordset object, it is also Read-only. If the Fields collection belongs to a Relation object, it is Not supported. If it belongs to a TableDef object, it is Read/write.

When a new record is created, the DefaultValue property setting is automatically entered as the value for the field. You can change the field value by setting its Value property.

The DefaultValue property doesn't apply to AutoNumber and Long Binary fields.
Description Property

Returns a descriptive string associated with an error. This is the default property for the Error object.

Return Values

The return value is a String data type that describes the error.

Remarks

The Description property comprises a short description of the error. Use this property to alert the user about an error that you cannot or do not want to handle.
DesignMasterID Property

Sets or returns a 16-byte value that uniquely identifies the Design Master in a replica set (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a GUID that uniquely identifies the Design Master.

Remarks

You should set the DesignMasterID property only if you need to move the current Design Master. Setting this property makes a specific replica in the replica set the Design Master.

Caution  Never create a second Design Master in a replica set. The existence of a second Design Master can result in the loss of data.

Under extreme circumstances — for example, if the Design Master is erased or corrupted — you can set this property at the current replica. However, setting this property at a replica when there is already another Design Master in the set might partition your replica set into two irreconcilable sets and prevent any further synchronization of data.
If you decide to make a replica the new Design Master for the set, synchronize it with all the replicas in the replica set before setting the **DesignMasterID** property in the replica. The replica must be open in exclusive mode in order to make it the Design Master.

If you make a replica that is designated read-only into the Design Master, the target replica is made read/write; the old Design Master also remains read/write.

The **DesignMasterID** property setting is stored in the MSysRepInfo system table.
Direction Property

Sets or returns a value that indicates whether a Parameter object represents an input parameter, an output parameter, both, or the return value from the procedure (ODBCDirect workspaces only).

Settings And Return Values

The setting or return value is a Long that can be set to one of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbParamInput</td>
<td>(Default) Passes information to the procedure.</td>
</tr>
<tr>
<td>dbParamInputOutput</td>
<td>Passes information both to and from the procedure.</td>
</tr>
<tr>
<td>dbParamOutput</td>
<td>Returns information from the procedure as in an output parameter in SQL.</td>
</tr>
<tr>
<td>dbParamReturnValue</td>
<td>Passes the return value from a procedure.</td>
</tr>
</tbody>
</table>

Remarks

Use the Direction property to determine whether the parameter is an input
parameter, output parameter, both, or the return value from the procedure. Some ODBC drivers do not provide information on the direction of parameters to a SELECT statement or procedure call. In these cases, it is necessary to set the direction prior to executing the query.

For example, the following procedure returns a value from a stored procedure named "get_employees":

```c
{? = call get_employees}
```

This call produces one parameter — the return value. You need to set the direction of this parameter to dbParamOutput or dbParamReturnValue before executing the QueryDef.

You need to set all parameter directions except dbParamInput before accessing or setting the values of the parameters and before executing the QueryDef.

You should use dbParamReturnValue for return values, but in cases where that option is not supported by the driver or the server, you can use dbParamOutput instead.

**Note** The Microsoft SQL Server 6.0 driver automatically sets the Direction property for all procedure parameters. Not all ODBC drivers can determine the direction of a query parameter. In these cases, it is necessary to set the direction prior to executing the query.
DistinctCount Property

Returns a value that indicates the number of unique values for the Index object that are included in the associated table (Microsoft Jet workspaces only).

Return Values

The return value is a Long data type.

Remarks

Check the DistinctCount property to determine the number of unique values, or keys, in an index. Any key is counted only once, even though there may be multiple occurrences of that value if the index permits duplicate values. This information is useful in applications that attempt to optimize data access by evaluating index information. The number of unique values is also known as the cardinality of an Index object.

The DistinctCount property won't always reflect the actual number of keys at a particular time. For example, a change caused by a rolled back transaction won't be reflected immediately in the DistinctCount property. The DistinctCount property value also may not reflect the deletion of records with unique keys. The number will be accurate immediately after you use the CreateIndex method.
EditMode Property

Returns a value that indicates the state of editing for the current record.

Return Values

The return value is a `Long` that indicates the state of editing, as listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbEditNone</td>
<td>No editing operation is in progress.</td>
</tr>
<tr>
<td>DbEditInProgress</td>
<td>The <code>Edit</code> method has been invoked, and the current record is in the copy buffer.</td>
</tr>
<tr>
<td></td>
<td>The <code>AddNew</code> method has been invoked, and the current record in the copy buffer is a new record that hasn't been saved in the database.</td>
</tr>
<tr>
<td>dbEditAdd</td>
<td></td>
</tr>
</tbody>
</table>

Remarks

The `EditMode` property is useful when an editing process is interrupted, for example, by an error during validation. You can use the value of the `EditMode` property to determine whether you should use the `Update` or `CancelUpdate`
method.

You can also check to see if the LockEdits property setting is True and the EditMode property setting is dbEditInProgress to determine whether the current page is locked.
FieldSize Property

Returns the number of bytes used in the database (rather than in memory) of a Memo or Long Binary Field object in the Fields collection of a Recordset object.

Return Values

The return value is a Long that indicates the number of characters (for a Memo field) or the number of bytes (for a Long Binary field).

Remarks

You can use FieldSize with the AppendChunk and GetChunk methods to manipulate large fields.

Because the size of a Long Binary or Memo field can exceed 64K, you should assign the value returned by FieldSize to a variable large enough to store a Long variable.

To determine the size of a Field object other than Memo and Long Binary types, use the Size property.

Note  In an ODBCDirect workspace, the FieldSize property is not available in
the following situations:

- If the database server or ODBC driver does not support server-side cursors.

- If you are using the ODBC cursor library (that is, the DefaultCursorDriver property is set to dbUseODBC, or to dbUseDefault when the server does not support server-side cursors).

- If you are using a cursorless query (that is, the DefaultCursorDriver property is set to dbUseNoCursor).

For example, Microsoft SQL Server version 4.21 does not support server-side cursors, so the FieldSize property is not available.

The FieldSize property and the VBA Len() or LenB() functions may return different values as the length of the same string. Strings are stored in a Microsoft Jet database in multi-byte character set (MBCS) form, but exposed through VBA in Unicode format. As a result, the Len() function will always return the number of characters, LenB will always return the number of characters X 2 (Unicode uses two bytes for each character), but FieldSize will return some value in between if the string has any MBCS characters. For example, given a string consisting of three normal characters and two MBCS characters, Len() will return 5, LenB() will return 10, and FieldSize will return 7, the sum of 1 for each normal character and 2 for each MBCS character.
Filter Property

Sets or returns a value that determines the records included in a subsequently opened Recordset object (Microsoft Jet workspaces only).

**Settings and Return Values**

The setting or return value is a String data type that contains the WHERE clause of an SQL statement without the reserved word WHERE.

**Remarks**

Use the Filter property to apply a filter to a dynaset-, snapshot-, or forward-only–type Recordset object.

You can use the Filter property to restrict the records returned from an existing object when a new Recordset object is opened based on an existing Recordset object.

In many cases, it's faster to open a new Recordset object by using an SQL statement that includes a WHERE clause.

Use the U.S. date format (month-day-year) when you filter fields containing
dates, even if you're not using the U.S. version of the Microsoft Jet database engine (in which case you must assemble any dates by concatenating strings, for example, strMonth & "-" & strDay & "-" & strYear). Otherwise, the data may not be filtered as you expect.

If you set the property to a string concatenated with a non-integer value, and the system parameters specify a non-U.S. decimal character such as a comma (for example, strFilter = "PRICE > " & lngPrice, and lngPrice = 125,50), an error occurs when you try to open the next Recordset. This is because during concatenation, the number will be converted to a string using your system's default decimal character, and Microsoft Jet SQL only accepts U.S. decimal characters.
Foreign Property

Returns a value that indicates whether an Index object represents a foreign key in a table (Microsoft Jet workspaces only).

Return Values

The return value is a Boolean data type that returns True if the Index object represents a foreign key.

Remarks

A foreign key consists of one or more fields in a foreign table that uniquely identify all rows in a primary table.

The Microsoft Jet database engine creates an Index object for the foreign table and sets the Foreign property when you create a relationship that enforces referential integrity.
ForeignName Property

Sets or returns a value that specifies the name of the Field object in a foreign table that corresponds to a field in a primary table for a relationship (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String data type that evaluates to the name of a Field in the associated TableDef object's Fields collection.

If the Relation object isn't appended to the Database, but the Field is appended to the Relation object, the ForeignName property is read/write. Once the Relation object is appended to the database, the ForeignName property is read-only.

Remarks

Only a Field object that belongs to the Fields collection of a Relation object can support the ForeignName property.

The Name and ForeignName property settings for a Field object specify the names of the corresponding fields in the primary and foreign tables of a
relationship. The **Table** and **ForeignTable** property settings for a **Relation** object determine the primary and foreign tables of a relationship.

For example, if you had a list of valid part codes (in a field named PartNo) stored in a ValidParts table, you could establish a relationship with an OrderItem table such that if a part code were entered into the OrderItem table, it would have to already exist in the ValidParts table. If the part code didn't exist in the ValidParts table and you had not set the **Attributes** property of the **Relation** object to **dbRelationDontEnforce**, a trappable error would occur.

In this case, the ValidParts table is the **foreign table**, so the **ForeignTable** property of the **Relation** object would be set to ValidParts and the **Table** property of the **Relation** object would be set to OrderItem. The **Name** and **ForeignName** properties of the **Field** object in the **Relation** object's **Fields** collection would be set to PartNo.

The following illustration depicts the relation described above.
ForeignTable Property

Sets or returns the name of the foreign table in a relationship (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String data type that evaluates to the name of a table in the Database object's TableDefs collection. This property is read/write for a new Relation object not yet appended to a collection and read-only for an existing Relation object in the Relations collection.

Remarks

The ForeignTable property setting of a Relation object is the Name property setting of the TableDef or QueryDef object that represents the foreign table or query; the Table property setting is the Name property setting of the TableDef or QueryDef object that represents the primary table or query.

For example, if you had a list of valid part codes (in a field named PartNo) stored in a ValidParts table, you could establish a relationship with an OrderItem table such that if a part code were entered into the OrderItem table, it would have to already be in the ValidParts table. If the part code didn't exist in the ValidParts
table and you had not set the Attributes property of the Relation object to dbRelationDontEnforce, a trappable error would occur.

In this case, the ValidParts table is the primary table, so the Table property of the Relation object would be set to ValidParts and the ForeignTable property of the Relation object would be set to OrderItem. The Name and ForeignName properties of the Field object in the Relation object's Fields collection would be set to PartNo.

The following illustration depicts the relation described above.
IgnoreNulls Property

Sets or returns a value that indicates whether records that have Null values in their index fields have index entries (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a Boolean that is True if the fields with Null values don't have an index entry. This property is read/write for a new Index object not yet appended to a collection and read-only for an existing Index object in an Indexes collection.

Remarks

To speed up the process of searching for records, you can define an index for a field. If you allow Null entries in an indexed field and expect many of the entries to be Null, you can set the IgnoreNulls property for the Index object to True to reduce the amount of storage space that the index uses.

The IgnoreNulls property setting and the Required property setting together determine whether a record with a Null index value has an index entry.

If IgnoreNulls is And Required is Then
<table>
<thead>
<tr>
<th>True</th>
<th>False</th>
<th>A <strong>Null</strong> value is allowed in the index field; no index entry added.</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>A <strong>Null</strong> value is allowed in the index field; index entry added.</td>
</tr>
<tr>
<td>True or False</td>
<td>True</td>
<td>A <strong>Null</strong> value isn't allowed in the index field; no index entry added.</td>
</tr>
</tbody>
</table>
Index Property

Sets or returns a value that indicates the name of the current Index object in a table-type Recordset object (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String data type that evaluates to the name of an Index object in the Indexes collection of the TableDef or table-type Recordset object's TableDef object.

Remarks

Records in base tables aren't stored in any particular order. Setting the Index property changes the order of records returned from the database; it doesn't affect the order in which the records are stored.

The specified Index object must already be defined. If you set the Index property to an Index object that doesn't exist or if the Index property isn't set when you use the Seek method, a trappable error occurs.

Examine the Indexes collection of a TableDef object to determine what Index objects are available to table-type Recordset objects created from that TableDef
object.

You can create a new index for the table by creating a new Index object, setting its properties, appending it to the Indexes collection of the underlying TableDef object, and then reopening the Recordset object.

Records returned from a table-type Recordset object can be ordered only by the indexes defined for the underlying TableDef object. To sort records in some other order, you can open a dynaset-, snapshot-, or forward-only--type Recordset object by using an SQL statement with an ORDER BY clause.

Notes

- You don't have to create indexes for tables. With large, unindexed tables, accessing a specific record or creating a Recordset object can take a long time. On the other hand, creating too many indexes slows down update, append, and delete operations because all indexes are automatically updated.

- Records read from tables without indexes are returned in no particular sequence.

- The Attributes property of each Field object in the Index object determines the order of records and consequently determines the access techniques to use for that index.

- A unique index helps optimize finding records.

- Indexes don't affect the physical order of a base table; indexes affect only how the records are accessed by the table-type Recordset object when a particular index is chosen or when Recordset is opened.
Inherit Property

Sets or returns a value that indicates whether new Document objects will inherit a default Permissions property setting (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a Boolean data type. If you set the property to True, Document objects inherit a default Permissions property setting.

Remarks

Use the Inherit property in conjunction with the Permissions property to define what permissions new documents will automatically have when they're created. If you set the Inherit property to True, and then set a permission on a container, then whenever a new document is created in that container, that permission will be set on the new document. This is a very convenient way of presetting permissions on an object.

Setting the Inherit property will not affect existing documents in the container? you can't modify all the permissions on all existing documents in a container by setting the Inherit property and a new permission. It will affect only new documents that are created after the Inherit property is set.
Inherited Property

Returns a value that indicates whether a Property object is inherited from an underlying object.

**Return Values**

The return value is a Boolean data type that is True if the Property object is inherited. For built-in Property objects that represent predefined properties, the only possible return value is False. This property is always False in an ODBCDirect workspace.

**Remarks**

You can use the Inherited property to determine whether a user-defined Property was created for the object it applies to, or whether the Property was inherited from another object. For example, suppose you create a new Property for a QueryDef object and then open a Recordset object from the QueryDef object. This new Property will be part of the Recordset object's Properties collection, and its Inherited property will be set to True because the property was created for the QueryDef object, not the Recordset object.
IniPath Property

Sets or returns information about the Windows Registry key that contains values for the Microsoft Jet database engine (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String data type that points to a user-supplied portion of the Windows Registry key containing Microsoft Jet database engine settings or parameters needed for installable ISAM databases.

Remarks

You can configure the Microsoft Jet engine with the Windows Registry. You can use the Registry to set options, such as installable ISAM DLLs.

For this option to have any effect, you must set the IniPath property before your application invokes any other DAO code. The scope of this setting is limited to your application and can't be changed without restarting your application.

You also use the Registry to provide initialization parameters for some installable ISAM database drivers. For example, to use Paradox version 4.0, set the IniPath property to a part of the Registry containing the appropriate
parameters.

This property recognizes either HKEY_LOCAL_MACHINE or HKEY_LOCAL_USER. If no root key is supplied, the default is HKEY_LOCAL_MACHINE.

Microsoft Jet versions 2.5 or earlier kept initialization information in .ini files.
IsolateODBCTrans Property

Sets or returns a value that indicates whether multiple transactions that involve the same Microsoft Jet-connected ODBC data source are isolated (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a Boolean data type that is True if you want to isolate transactions involving the same ODBC (Open Database Connectivity) connection. False (the default) will allow multiple transactions involving the same ODBC connection.

Remarks

In some situations, you need to have multiple simultaneous transactions pending on the same ODBC connection. To do this, you need to open a separate Workspace for each transaction. Although each Workspace can have its own ODBC connection to the database, this slows system performance. Because transaction isolation isn't usually required, ODBC connections from multiple Workspace objects opened by the same user are shared by default.

Some ODBC servers, such as Microsoft SQL Server, don't allow simultaneous
transactions on a single connection. If you need to have more than one transaction at a time pending against such a database, set the **IsolateODBCTrans** property to **True** on each **Workspace** as soon as you open it. This forces a separate ODBC connection for each **Workspace**.
KeepLocal Property

Sets or returns a value on a table, query, form, report, macro, or module that you do not want to replicate when the database is replicated (Microsoft Jet workspaces only).

**Note** Before getting or setting the KeepLocal property on a TableDef, or QueryDef object, you must create it by using the CreateProperty method and append it to the Properties collection for the object.

**Settings and Return Values**

The setting or return value is a Text data type. If you set this property to "T", the object will remain local when the database is replicated. You can't use the KeepLocal property on objects after they have been replicated.

**Remarks**

Once you set the KeepLocal property, it will appear in the Properties collection for the Document object representing the host object.

Before setting the KeepLocal property, you should check the value of the Replicable property.
After you make a database replicable, all new objects created within the Design Master, or in any other replicas in the set, are local objects. Local objects remain in the replica in which they're created and aren't copied throughout the replica set. Each time you make a new replica in the set, the new replica contains all the replicable objects from the source replica, but none of the local objects from the source replica.

If you create a new object in a replica and want to change it from local to replicable so that all users can use it, you can either create the object in or import it into the Design Master. Be sure to delete the local object from any replicas; otherwise, you will encounter a design error. After the object is part of the Design Master, set the object's Replicable property to True.

The object on which you are setting the KeepLocal property might have already inherited that property from another object. However, the value set by the other object has no effect on the behavior of the object you want to keep local. You must explicitly set the property for each object.
LastModified Property

Returns a bookmark indicating the most recently added or changed record.

**Return Values**

The return value is a Variant array of Byte data.

**Remarks**

You can use the LastModified property to move to the most recently added or updated record. Use the LastModified property with table- and dynaset-type Recordset objects. A record must be added or modified in the Recordset object itself in order for the LastModified property to have a value.
LockEdits Property

Sets or returns a value indicating the type of locking that is in effect while editing.

**Settings and Return Values**

The setting or return value is a Boolean that indicates the type of locking, as specified in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>Default. Pessimistic locking is in effect. The 2K page containing the record you're editing is locked as soon as you call the Edit method. Optimistic locking is in effect for editing. The 2K page containing the record is not locked until the Update method is executed.</td>
</tr>
<tr>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

**Remarks**

You can use the LockEdits property with updatable Recordset objects.
If a page is locked, no other user can edit records on the same page. If you set \textbf{LockEdits} to \textbf{True} and another user already has the page locked, an error occurs when you use the \textbf{Edit} method. Other users can read data from locked pages.

If you set the \textbf{LockEdits} property to \textbf{False} and later use the \textbf{Update} method while another user has the page locked, an error occurs. To see the changes made to your record by another user, use the \textbf{Move} method with 0 as the argument; however, if you do this, you will lose your changes.

When working with \textbf{Microsoft Jet-connected ODBC data sources}, the \textbf{LockEdits} property is always set to \textbf{False}, or optimistic locking. The \textbf{Microsoft Jet database engine} has no control over the locking mechanisms used in external database servers.

\textbf{Note}  You can preset the value of \textbf{LockEdits} when you first open the \textbf{Recordset} by setting the \textit{lockedits} argument of the \textbf{OpenRecordset} method. Setting the \textit{lockedits} argument to \textbf{dbPessimistic} will set the \textbf{LockEdits} property to \textbf{True}, and setting \textit{lockedits} to any other value will set the \textbf{LockEdits} property to \textbf{False}. 
LoginTimeout Property

Sets or returns the number of seconds before an error occurs when you attempt to log on to an ODBC database.

Settings and Return Values

The setting or return value is an Integer representing the number of seconds before a login timeout error occurs. The default LoginTimeout property setting is 20 seconds. When the LoginTimeout property is set to 0, no timeout occurs.

Remarks

When you're attempting to log on to an ODBC database, such as Microsoft SQL Server, the connection can fail as a result of network errors or because the server isn't running. Rather than waiting for the default 20 seconds to connect, you can specify how long to wait before raising an error. Logging on to the server happens implicitly as part of a number of different events, such as running a query on an external server database.

You can use LoginTimeout on the DBEngine object in both Microsoft Jet and ODBCDirect workspaces. You can use LoginTimeout on the Workspace object only in ODBCDirect workspaces. Setting the property to -1 on a Workspace
will default to the current setting of **DBEngine.LoginTimeout**. You can change this property in a **Workspace** at any time, and the new setting will take effect with the next **Connection** or **Database** object opened.

The default value is determined by the ODBC driver. In a Microsoft Jet workspace, you can override the driver’s default value by creating a new “ODBC” key in the Registry path `\HKEY_LOCAL_MACHINE\SOFTWARE\Jet\3.5`, creating a **LoginTimeout** parameter in this key, and setting the value as desired.
LogMessages Property

Sets or returns a value that specifies if the messages returned from a Microsoft Jet-connected ODBC data source are recorded (Microsoft Jet workspaces only).

**Note** Before you can set or get the value of the LogMessages property, you must create the LogMessages property with the CreateProperty method, and append it to the Properties collection of a QueryDef object.

**Settings and Return Values**

The setting or return value is a **Boolean** that is **True** if ODBC-generated messages are recorded.

**Remarks**

Some pass-through queries can return messages in addition to data. If you set the LogMessages property to **True**, the Microsoft Jet database engine creates a table that contains returned messages. The table name is the user name concatenated with a hyphen (-) and a sequential number starting at 00. For example, because the default user name is Admin, the tables returned would be named Admin-00, Admin-01, and so on.
If you expect the query to return messages, create and append a user-defined LogMessages property for the QueryDef object, and set its type to Boolean and its value to True.

Once you've processed the results from these tables, you may want to delete them from the database along with the temporary query used to create them.
MaxRecords Property

Sets or returns the maximum number of records to return from a query against an ODBC data source.

Settings And Return Values

The setting or return value is a Long that represents the number of records to be returned. The default value is 0, indicating no limit on the number of records returned.

Remarks

Once the number of rows specified by MaxRecords is returned to your application in a Recordset, the query processor will stop returning additional records even if more records would qualify for inclusion in the Recordset. This property is useful in situations where limited client resources prohibit management of large numbers of records.

Note  The MaxRecords property can only be used with an ODBC data source, connected through either a Microsoft Jet or ODBCDirect workspace.
Name Property

Sets or returns a user-defined name for a DAO object. For an object not appended to a collection, this property is read/write.

**Settings and Return Values**

The setting or return value is a String that specifies a name. The name must start with a letter. The maximum number of characters depends on the type of object Name applies to, as shown in Remarks. It can include numbers and underscore characters ( _) but can't include punctuation or spaces.

**Remarks**

TableRow, QueryDef, Field, Index, User, and Group objects can't share the same name with any object in the same collection.

The Name property of a Recordset object opened by using an SQL statement is the first 256 characters of the SQL statement.

You can use an object's Name property with the Visual Basic for Applications Dim statement in code to create other instances of the object.
For many of the DAO objects, the **Name** property reflects the name as known to the **Database** object, as in the name of a **TableDef**, **Field**, or **QueryDef** object. There is no direct link between the name of the DAO object and the object variable used to reference it.

The read/write usage of the **Name** property depends on the type of object it applies to, and whether or not the object has been appended to a collection. In an **ODBCDirect workspace**, the **Name** property of an appended object is always read-only. The following table indicates whether the **Name** property in a **Microsoft Jet workspace** is read/write or read-only for an object that is appended to a collection (unless otherwise noted), and also indicates its maximum length in cases where it is read/write.

<table>
<thead>
<tr>
<th>Object</th>
<th>Usage</th>
<th>Maximum length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Connection</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Document</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>Field</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unappended</td>
<td>Read/write</td>
<td>64</td>
</tr>
<tr>
<td>Appended to <strong>Index</strong></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Appended to <strong>QueryDef</strong></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Appended to <strong>Recordset</strong></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Appended to <strong>TableDef</strong> (native)</td>
<td>Read/write</td>
<td>64</td>
</tr>
<tr>
<td>Appended to <strong>TableDef</strong> (linked)</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Appended to <strong>Relation</strong></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>Group</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unappended</td>
<td>Read/write</td>
<td>20</td>
</tr>
<tr>
<td>Appended</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unappended</td>
<td>Read/write</td>
<td>64</td>
</tr>
<tr>
<td>Appended</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unappended</td>
<td>Read/write</td>
<td>64</td>
</tr>
<tr>
<td>Appended</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>Built-in</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>QueryDef</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unappended</td>
<td>Read/write</td>
<td></td>
</tr>
<tr>
<td>Temporary</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Appended</td>
<td>Read/write</td>
<td></td>
</tr>
<tr>
<td><strong>Recordset</strong></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>Relation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unappended</td>
<td>Read/write</td>
<td></td>
</tr>
<tr>
<td>Appended</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>TableDef</strong></td>
<td>Read/write</td>
<td></td>
</tr>
<tr>
<td><strong>User</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unappended</td>
<td>Read/write</td>
<td></td>
</tr>
<tr>
<td>Appended</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>Workspace</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unappended</td>
<td>Read/write</td>
<td></td>
</tr>
<tr>
<td>Appended</td>
<td>Read-only</td>
<td></td>
</tr>
</tbody>
</table>
NoMatch Property

Indicates whether a particular record was found by using the Seek method or one of the Find methods (Microsoft Jet workspaces only).

Return Values

The return value is a Boolean that is True if the desired record was not found. When you open or create a Recordset object, its NoMatch property is set to False.

Remarks

To locate a record, use the Seek method on a table-type Recordset object or one of the Find methods on a dynaset- or snapshot-type Recordset object. Check the NoMatch property setting to see whether the record was found.

If the Seek or Find method is unsuccessful and the NoMatch property is True, the current record will no longer be valid. Be sure to obtain the current record's bookmark before using the Seek method or a Find method if you'll need to return to that record.

Note  Using any of the Move methods on a Recordset object won't affect its
NoMatch property setting.
Number Property

Returns a numeric value specifying an error.

**Return Values**

The return value is a [Long](#) data type that represents an error number.

**Remarks**

Use the **Number** property to determine the error that occurred. The value of the property corresponds to a unique trap number that corresponds to an error condition. For a complete list of all trap numbers and error conditions, see [Trappable Microsoft Jet Errors](#).
ODBCTimeout Property

Indicates the number of seconds to wait before a timeout error occurs when a QueryDef is executed on an ODBC database.

**Settings and Return Values**

The setting or return value is an Integer representing the number of seconds to wait before a timeout error occurs.

When the ODBCTimeout property is set to -1, the timeout defaults to the current setting of the QueryTimeout property of the Connection or Database object that contains the QueryDef. When the ODBCTimeout property is set to 0, no timeout error occurs.

**Remarks**

When you're using an ODBC database, such as Microsoft SQL Server, delays can occur because of network traffic or heavy use of the ODBC server. Rather than waiting indefinitely, you can specify how long to wait before returning an error.

Setting the ODBCTimeout property of a QueryDef object overrides the value
specified by the **QueryTimeout** property of the **Connection** or **Database** object containing the **QueryDef**, but only for that **QueryDef** object.

**Note** In an ODBC Direct workspace, after setting **ODBCTimeout** to an explicit value you can reset it back to the default (i.e., -1) only once during the life of the **QueryDef** object. Otherwise, an error will occur.
OrdinalPosition Property

Sets or returns the relative position of a Field object within a Fields collection. For an object not yet appended to the Fields collection, this property is read/write.

**Settings and Return Values**

The setting or return value is an Integer that specifies the numeric order of fields. The default is 0.

**Remarks**

The availability of the OrdinalPosition property depends on the object that contains the Fields collection, as shown in the following table.

<table>
<thead>
<tr>
<th>If the Fields collection belongs to a</th>
<th>Then OrdinalPosition is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index object</td>
<td>Not supported</td>
</tr>
<tr>
<td>QueryDef object</td>
<td>Read-only</td>
</tr>
<tr>
<td>Recordset object</td>
<td>Read-only</td>
</tr>
<tr>
<td>Relation object</td>
<td>Not supported</td>
</tr>
</tbody>
</table>
**TableDef** object Read/write

Generally, the ordinal position of an object that you append to a collection depends on the order in which you append the object. The first appended object is in the first position (0), the second appended object is in the second position (1), and so on. The last appended object is in ordinal position $count - 1$, where $count$ is the number of objects in the collection as specified by the **Count** property setting.

You can use the **OrdinalPosition** property to specify an ordinal position for new **Field** objects that differs from the order in which you append those objects to a collection. This enables you to specify a field order for your tables, queries, and recordsets when you use them in an application. For example, the order in which fields are returned in a **SELECT *** query is determined by the current **OrdinalPosition** property values.

You can permanently reset the order in which fields are returned in recordsets by setting the **OrdinalPosition** property to any positive integer.

Two or more **Field** objects in the same collection can have the same **OrdinalPosition** property value, in which case they will be ordered alphabetically. For example, if you have a field named Age set to 4 and you set a second field named Weight to 4, Weight is returned after Age.

You can specify a number that is greater than the number of fields minus 1. The field will be returned in an order relative to the largest number. For example, if you set a field's **OrdinalPosition** property to 20 (and there are only 5 fields) and you've set the **OrdinalPosition** property for two other fields to 10 and 30, respectively, the field set to 20 is returned between the fields set to 10 and 30.

**Note**  Even if the **Fields** collection of a **TableDef** has not been refreshed, the field order in a **Recordset** opened from the **TableDef** will reflect the **OrdinalPosition** data of the **TableDef** object. A table-type **Recordset** will have the same **OrdinalPosition** data as the underlying table, but any other type of **Recordset** will have new **OrdinalPosition** data (starting with 0) that follow the order determined by the **OrdinalPosition** data of the **TableDef**.
Returns the value of a **Field** in the database that existed when the last **batch update** began (**ODBCDirect workspaces** only).

**Return Values**

The return value is a **variant** expression.

**Remarks**

During an optimistic batch update, a **collision** may occur where a second client modifies the same field and record in between the time the first client retrieves the data and the first client's update attempt. The **OriginalValue** property contains the value of the field at the time the last batch **Update** began. If this value does not match the value actually in the database when the batch **Update** attempts to write to the database, a collision occurs. When this happens, the new value in the database will be accessible through the **VisibleValue** property.
Owner Property

Sets or returns a value that specifies the owner of the object (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String that evaluates to either the name of a User object in the Users collection or the name of a Group object in the Groups collection.

Remarks

The owner of an object has certain access privileges denied to other users. Any individual user account (represented by a User object) or group of user accounts (represented by a Group object) can change the Owner property setting at any time if it has the appropriate permissions.
PartialReplica Property

Sets or returns a value on a Relation object indicating whether that relation should be considered when populating a partial replica from a full replica. (Microsoft Jet databases only.)

Settings and Return Values

The setting or return value is a Boolean data type that is True when the relation should be enforced during synchronization.

Remarks

This property enables you to replicate data from the full replica to the partial replica based on relationships between tables. You can use the PartialReplica property when setting the ReplicaFilter property alone can't adequately specify what data should be replicated to the partial. For example, suppose you have a database in which the Customers table has a one-to-many relationship with the Orders table, and you want to configure a partial replica that only replicates orders from customers in the California region (instead of all orders). It is not possible to set the ReplicaFilter property on the Orders table to Region = 'CA' because the Region field is in the Customers table, not the Orders table.
To replicate all orders from the California region, you must indicate that the relation between the Orders and Customers tables will be active during replication. Once you've created a partial replica, the following steps will populate it with all orders from the California region:

1. Set the **ReplicaFilter** property on the Customers **TableDef** object to "Region = 'CA'".

2. Set the value of the **PartialReplica** property to **True** on the **Relation** object corresponding to the relationship between Orders and Customers.

3. Invoke the **PopulatePartial** method.

   **Caution** When you set a replica filter or replica relation, be aware that records in the partial replica that don't satisfy the restriction criteria will be removed from the partial replica, but not from the full replica. For example, suppose you set the **ReplicaFilter** property on the Customers **TableDef** in the partial replica to "Region = 'CA'" and you then repopulate the database. This will insert or update all records for California-based customers. If you then reset the **ReplicaFilter** property to "Region = 'FL'" and repopulate the database, all California region records in the partial replica will be removed, and all records from Florida-based customers will be inserted from the full replica. No records in the full replica will be deleted.

Before setting either the **ReplicaFilter** or **PartialReplica** property, it's a good idea to synchronize the partial replica in which you are setting these properties with the full replica. This will ensure that pending changes in the partial replica will be merged into the full replica before any records are removed in the partial replica.
Password Property

Sets the password for a user account (Microsoft Jet workspaces only).

**Settings**

The setting is a String that can be up to 14 characters long and can include any characters except the ASCII character 0 (null). This property setting is write-only for new objects not yet appended to a collection, and is not available for existing objects.

**Remarks**

Set the **Password** property along with the **PID** property when you create a new **User** object.

Use the **NewPassword** method to change the **Password** property setting for an existing **User** object. To clear a password, set the *newpassword* argument of the **NewPassword** method to a zero-length string ("").

Passwords are case-sensitive.

**Note**  If you don't have access permission, you can't change the password of any
other user.
PercentPosition Property

Sets or returns a value indicating the approximate location of the current record in the Recordset object based on a percentage of the records in the Recordset.

Settings and Return Values

The setting or return value is a Single that is a number between 0.0 and 100.00.

Remarks

To indicate or change the approximate position of the current record in a Recordset object, you can check or set the PercentPosition property. When working with a dynaset- or snapshot-type Recordset object opened directly from a base table, first populate the Recordset object by moving to the last record before you set or check the PercentPosition property. If you use the PercentPosition property before fully populating the Recordset object, the amount of movement is relative to the number of records accessed as indicated by the RecordCount property setting. You can move to the last record by using the MoveLast method.

Note  Using the PercentPosition property to move the current record to a specific record in a Recordset object isn't recommended?the Bookmark property
is better suited for this task.

Once you set the **PercentPosition** property to a value, the record at the approximate position corresponding to that value becomes current, and the **PercentPosition** property is reset to a value that reflects the approximate position of the current record. For example, if your **Recordset** object contains only five records, and you set its **PercentPosition** property value to 77, the value returned from the **PercentPosition** property may be 80, not 77.

The **PercentPosition** property applies to all types of **Recordset** objects except for forward-only–type **Recordset** objects or **Recordset** objects opened from pass-through queries against remote databases.

You can use the **PercentPosition** property with a scroll bar on a form or text box to indicate the location of the current record in a **Recordset** object.
Permissions Property

Sets or returns a value that establishes the permissions for the user or group identified by the UserName property of a Container or Document object (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a Long constant that establishes permissions. The following tables list the valid constants for the Permissions property of various DAO objects. Unless otherwise noted, all constants shown in all tables are valid for Document objects.

The following table lists possible values for Container objects other than Tables and Databases containers.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecNoAccess</td>
<td>The user doesn't have access to the object (not valid for Document objects).</td>
</tr>
<tr>
<td>dbSecFullAccess</td>
<td>The user has full access to the object.</td>
</tr>
<tr>
<td>dbSecDelete</td>
<td>The user can delete the object.</td>
</tr>
</tbody>
</table>
The user can read the object's security-related information.

The user can alter access permissions.

The user can change the Owner property setting.

The following table lists the possible settings and return values for the Tables container.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecCreate</td>
<td>The user can create new documents (not valid for Document objects).</td>
</tr>
<tr>
<td>dbSecReadDef</td>
<td>The user can read the table definition, including column and index information.</td>
</tr>
<tr>
<td>dbSecWriteDef</td>
<td>The user can modify or delete the table definition, including column and index information.</td>
</tr>
<tr>
<td>dbSecRetrieveData</td>
<td>The user can retrieve data from the Document object.</td>
</tr>
<tr>
<td>dbSecInsertData</td>
<td>The user can add records.</td>
</tr>
<tr>
<td>dbSecReplaceData</td>
<td>The user can modify records.</td>
</tr>
<tr>
<td>dbSecDeleteData</td>
<td>The user can delete records.</td>
</tr>
</tbody>
</table>

The following table lists the possible settings and return values for the Databases container.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbSecDBAdmin</td>
<td>The user can replicate a database and change the database password (not valid for Document objects).</td>
</tr>
<tr>
<td>dbSecDBCreate</td>
<td>The user can create new databases. This option is valid only on the Databases container in the workgroup information file (Systen.mdw). This constant isn't valid for Document</td>
</tr>
</tbody>
</table>
**Remarks**

Use this property to establish or determine the type of read/write permissions the user has for a **Container** or **Document** object.

A **Document** object inherits the permissions for users from its **Container** object, provided the **Inherit** property of the **Container** object is set for those users or for a group to which the users belong. By setting a **Document** object's **Permissions** and **UserName** properties later, you can further refine the access control behavior of your object.

If you want to set or return permissions for a user that includes permissions inherited from any groups to which the user belongs, use the **AllPermissions** property.
PID Property

Sets the personal identifier (PID) for either a group or a user account (Microsoft Jet workspaces only).

Settings

The setting is a String containing 4-20 alphanumeric characters. This property setting is write-only for new objects not yet appended to a collection, and is not available for existing objects.

Remarks

Set the PID property along with the Name property when you create a new Group object. Set the PID property along with the Name and Password properties when you create a new User object.
Prepare Property

Sets or returns a value that indicates whether the query should be prepared on the server as a temporary stored procedure, using the ODBC SQLPrepare API function, prior to execution, or just executed using the ODBC SQLExecDirect API function (ODBCDirect workspaces only).

Settings and Return Values

The setting or return value is a Long value that can be one of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbQPrepare</td>
<td>(Default) The statement is prepared (that is, the ODBC SQLPrepare API is called).</td>
</tr>
<tr>
<td>dbQUnprepare</td>
<td>The statement is not prepared (that is, the ODBC SQLExecDirect API is called).</td>
</tr>
</tbody>
</table>

Remarks

You can use the Prepare property to either have the server create a temporary stored procedure from your query and then execute it, or just have the query
executed directly. By default the **Prepare** property is set to **dbQPrepare**. However, you can set this property to **dbQUnprepare** to prohibit preparing of the query. In this case, the query is executed using the **SQLExecDirect** API.

Creating a stored procedure can slow down the initial operation, but increases performance of all subsequent references to the query. However, some queries cannot be executed in the form of stored procedures. In these cases, you must set the **Prepare** property to **dbQUnprepare**.

If **Prepare** is set to **dbQPrepare**, this can be overridden when the query is executed by setting the **Execute** method's **options** argument to **dbExecDirect**.

**Note** The ODBC **SQLPrepare** API is called as soon as the DAO **SQL** property is set. Therefore, if you want to improve performance using the **dbQUnprepare** option, you must set the **Prepare** property before setting the **SQL** property.
Sets or returns a value that indicates whether an Index object represents a primary key index for a table (Microsoft Jet workspaces only).

**Settings and Return Values**

The setting or return value is a Boolean that is True if the Index object represents a primary key index.

The Primary property setting is read/write for a new Index object not yet appended to a collection and read-only for an existing Index object in an Indexes collection. If the Index object is appended to the TableDef object but the TableDef object isn't appended to the TableDefs collection, the Index property is read/write.

**Remarks**

A primary key index consists of one or more fields that uniquely identify all records in a table in a predefined order. Because the index field must be unique, the Unique property of the Index object is set to True. If the primary key index consists of more than one field, each field can contain duplicate values, but each combination of values from all the indexed fields must be unique. A primary key
index consists of a key for the table and usually contains the same fields as the primary key.

Note  You don't have to create indexes for tables, but in large, unindexed tables, accessing a specific record can take a long time. The Attributes property of each Field object in the Index object determines the order of records and consequently determines the access techniques to use for that index. When you create a new table in your database, it's a good idea to create an index on one or more fields that uniquely identify each record, and then set the Primary property of the Index object to True.

When you set a primary key for a table, the primary key is automatically defined as the primary key index for the table.
QueryTimeout Property

Sets or returns a value that specifies the number of seconds to wait before a timeout error occurs when a query is executed on an ODBC data source.

Settings and Return Values

The setting or return value is an Integer representing the number of seconds to wait. The default value is 60.

Remarks

When you're using an ODBC database, such as Microsoft SQL Server, there may be delays due to network traffic or heavy use of the ODBC server. Rather than waiting indefinitely, you can specify how long to wait.

When you use QueryTimeout with a Connection or Database object, it specifies a global value for all queries associated with the database. You can override this value for a specific query by setting the ODBCTimeout property of the particular QueryDef object.

In a Microsoft Jet workspace, you can override the default value by creating a new “ODBC” key in the Registry path.
\HKEY_LOCAL_MACHINE\SOFTWARE\Jet\3.5\, creating a QueryTimeout parameter in this key, and setting the value as desired.
RecordCount Property

Returns the number of records accessed in a Recordset object, or the total number of records in a table-type Recordset or TableDef object.

Return Values

The return value is a Long data type.

Remarks

Use the RecordCount property to find out how many records in a Recordset or TableDef object have been accessed. The RecordCount property doesn't indicate how many records are contained in a dynaset-, snapshot-, or forward-only–type Recordset object until all records have been accessed. Once the last record has been accessed, the RecordCount property indicates the total number of undeleted records in the Recordset or TableDef object. To force the last record to be accessed, use the MoveLast method on the Recordset object. You can also use an SQL Count function to determine the approximate number of records your query will return.

Note   Using the MoveLast method to populate a newly opened Recordset negatively impacts performance. Unless it is necessary to have an accurate
RecordCount as soon as you open a Recordset, it's better to wait until you populate the Recordset with other portions of code before checking the RecordCount property.

As your application deletes records in a dynaset-type Recordset object, the value of the RecordCount property decreases. However, records deleted by other users aren't reflected by the RecordCount property until the current record is positioned to a deleted record. If you execute a transaction that affects the RecordCount property setting and you subsequently roll back the transaction, the RecordCount property won't reflect the actual number of remaining records.

The RecordCount property of a snapshot- or forward-only–type Recordset object isn't affected by changes in the underlying tables.

A Recordset or TableDef object with no records has a RecordCount property setting of 0.

When you work with linked TableDef objects, the RecordCount property setting is always –1.

Using the Requery method on a Recordset object resets the RecordCount property just as if the query were re-executed.
RecordsAffected Property

Returns the number of records affected by the most recently invoked **Execute** method.

**Return Values**

The return value is a **Long** from 0 to the number of records affected by the most recently invoked **Execute** method on either a **Database** or **QueryDef** object.

**Remarks**

When you use the **Execute** method to run an **action query** from a **QueryDef** object, the **RecordsAffected** property will contain the number of records deleted, updated, or inserted.

When you use **RecordsAffected** in an **ODBCDirect workspace**, it will not return a useful value from an SQL DROP TABLE action query.
RecordStatus Property

Returns a value indicating the update status of the current record if it is part of a batch update (ODBCDirect workspaces only).

Return Values

The return value is a Long that can be any of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbRecordUnmodified</td>
<td>(Default) The record has not been modified or has been updated successfully.</td>
</tr>
<tr>
<td>dbRecordModified</td>
<td>The record has been modified and not updated in the database.</td>
</tr>
<tr>
<td>dbRecordNew</td>
<td>The record has been inserted with the AddNew method, but not yet inserted into the database.</td>
</tr>
<tr>
<td>dbRecordDeleted</td>
<td>The record has been deleted, but not yet deleted in the database.</td>
</tr>
<tr>
<td>dbRecordDBDeleted</td>
<td>The record has been deleted locally and in the database.</td>
</tr>
</tbody>
</table>
Remarks

The value of the **RecordStatus** property indicates whether and how the current record will be involved in the next optimistic batch update.

When a user changes a record, the **RecordStatus** for that record automatically changes to **dbRecordModified**. Similarly, if a record is added or deleted, **RecordStatus** reflects the appropriate constant. When you then use a **batch-mode Update** method, DAO will submit an appropriate operation to the remote server for each record, based on the record's **RecordStatus** property.
Replicable Property

Sets or returns a value that determines whether a database or object in a database can be replicated (Microsoft Jet workspaces only).

**Note** Before getting or setting the **Replicable** property on a **Database**, **TableDef**, or **QueryDef** object, you must create it by using the **CreateProperty** method and append it to the **Properties** collection for the object.

**Setting and Return Values**

The setting or return value is a **Text** data type.

On a **Database** object, setting this property to "T" makes the database replicable. Once you set the property to "T", you can't change it; setting the property to "F" (or any value other than "T") causes an error.

On an object in a database, setting this property to "T" replicates the object (and subsequent changes to the object) at all **replicas** in the **replica set**. You can also set this property in the object's property sheet in Microsoft Access.

**Note** Microsoft Jet 3.5 also supports the **Boolean ReplicableBool** property. Its functionality is identical to the **Replicable** property, except that it takes a
**Boolean** value. Setting **ReplicableBool** to **True** makes the object replicable.

**Remarks**

Before setting the **Replicable** property on a database, make a backup copy of the database. If setting the **Replicable** property fails, you should delete the partially replicated database, make a new copy from the backup, and try again.

When you set this property on a **Database** object, Microsoft Jet adds fields, tables, and properties to objects within the database. Microsoft Jet uses these fields, tables, and properties to synchronize database objects. For example, all existing tables have three new fields added to them that help identify which records have changed. The addition of these fields and other objects increase the size of your database.

On forms, reports, macros, and modules defined by a host application (such as Microsoft Access), you set this property on the host-defined object through the host user interface. Once set, the **Replicable** property will appear in the **Properties** collection for the **Document** object representing the host object.

If the **Replicable** property has already been set on an object using the **Replicated** check box in the property sheet for the object, you cannot set the **Replicable** property in code.

When you create a new table, query, form, report, macro, or module at a replica, the object is considered local and is stored only at that replica. If you want users at other replicas to be able to use the object, you must change it from local to replicable. Either create the object at or import it into the **Design Master** and then set the **Replicable** property to "T".

The object on which you are setting the **Replicable** property might have already inherited that property from another object. However, the value set by the other object has no effect on the behavior of the object you want to make replicable. You must explicitly set the property for each object.
ReplicaFilter Property

Sets or returns a value on a TableDef object within a partial replica that indicates which subset of records is replicated to that table from a full replica. (Microsoft Jet databases only.)

Settings And Return Values

The setting or return value is a String or Boolean that indicates which subset of records is replicated, as specified in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A string</td>
<td>A criteria that a record in the partial replica table must satisfy in order to be replicated from the full replica.</td>
</tr>
<tr>
<td>True</td>
<td>Replicates all records.</td>
</tr>
<tr>
<td>False</td>
<td>(Default) Doesn't replicate any records.</td>
</tr>
</tbody>
</table>

Remarks

This property is similar to an SQL WHERE clause (without the word WHERE), but you cannot specify subqueries, aggregate functions (such as Count), or user-
defined functions within the criteria.

You can only synchronize data between a full replica and a partial replica. You can't synchronize data between two partial replicas. Also, with partial replication you can set restrictions on which records are replicated, but you can't indicate which fields are replicated.

Usually, you reset a replica filter when you want to replicate a different set of records. For example, when a sales representative temporarily takes over another sales representative's region, the database application can temporarily replicate data for both regions and then return to the previous filter. In this scenario, the application resets the **ReplicaFilter** property and then repopulates the partial replica.

If your application changes replica filters, you should follow these steps:

1. Use the **Synchronize** method to synchronize your full replica with the partial replica in which the filters are being changed.

2. Use the **ReplicaFilter** property to make the desired changes to the replica filter.

3. Use the **PopulatePartial** method to remove all records from the partial replica and transfer all records from the full replica that meet the new replica filter criteria.

To remove a filter, set the **ReplicaFilter** property to **False**. If you remove all filters and invoke the **PopulatePartial** method, no records will appear in any replicated tables in the partial replica.

**Note** If a replica filter has changed, and the **Synchronize** method is invoked without first invoking **PopulatePartial**, a trappable error occurs.
ReplicaID Property

Returns a 16-byte value that uniquely identifies a database replica (Microsoft Jet workspaces only).

Return Values

The return value is a GUID value that uniquely identifies the replica or Design Master.

Remarks

The Microsoft Jet database engine automatically generates this value when you create a new replica.

The ReplicaID property of each replica (and the Design Master) is stored in the MSysReplicas system table.
Sets or returns a value that indicates whether a Field object requires a non-Null value or whether all the fields in an Index object must have a value.

**Settings and Return Values**

The setting or return value is a Boolean that is True if a field can't contain a Null value.

For an object not yet appended to a collection, this property is read/write. For an Index object, this property setting is read-only for objects appended to Indexes collections in Recordset and TableDef objects.

**Remarks**

The availability of the Required property depends on the object that contains the Fields collection, as shown in the following table.

<table>
<thead>
<tr>
<th>If the Fields collection belongs to a</th>
<th>Then Required is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index object</td>
<td>Not supported</td>
</tr>
<tr>
<td>QueryDef object</td>
<td>Read-only</td>
</tr>
</tbody>
</table>
**Recordset** object  
Read-only  

**Relation** object  
Not supported  

**TableDef** object  
Read/write  

For a **Field** object, you can use the **Required** property along with the **AllowZeroLength**, **ValidateOnSet**, or **ValidationRule** property to determine the validity of the **Value** property setting for that **Field** object. If the **Required** property is set to **False**, the field can contain **Null** values as well as values that meet the conditions specified by the **AllowZeroLength** and **ValidationRule** property settings.

**Note**  
When you can set this property for either an **Index** object or a **Field** object, set it for the **Field** object. The validity of the property setting for a **Field** object is checked before that of an **Index** object.
Restartable Property

Returns a value that indicates whether a Recordset object supports the Requery method, which re-executes the query on which the Recordset object is based.

Return Values

The return value is a Boolean data type that is True if the Recordset object supports the Requery method. Table-type Recordset objects always return False.

Remarks

Check the Restartable property before using the Requery method on a Recordset object. If the object's Restartable property is set to False, use the OpenRecordset method on the underlying QueryDef object to re-execute the query.
ReturnsRecords Property

Sets or returns a value that indicates whether an SQL pass-through query to an external database returns records (Microsoft Jet workspaces only).

**Settings and Return Values**

The setting or return value is a Boolean that is True (default) if a pass-through query returns records.

**Remarks**

Not all SQL pass-through queries to external databases return records. For example, an SQL UPDATE statement updates records without returning records, while an SQL SELECT statement does return records. If the query returns records, set the ReturnsRecords property to True; if the query doesn't return records, set the ReturnsRecords property to False.

**Note** You must set the Connect property before you set the ReturnsRecords property.
Size Property

Sets or returns a value that indicates the maximum size, in bytes, of a Field object.

Settings and Return Values

The setting or return value is a constant that indicates the maximum size of a Field object. For an object not yet appended to the Fields collection, this property is read/write. The setting depends on the Type property setting of the Field object, as discussed under Remarks.

Remarks

For fields (other than Memo type fields) that contain character data, the Size property indicates the maximum number of characters that the field can hold. For numeric fields, the Size property indicates how many bytes of storage are required.

Use of the Size property depends on the object that contains the Fields collection to which the Field object is appended, as shown in the following table.

<table>
<thead>
<tr>
<th>Object appended to</th>
<th>Usage</th>
</tr>
</thead>
</table>
Index: Not supported  
QueryDef: Read-only  
Recordset: Read-only  
Relation: Not supported  
TableDef: Read-only

When you create a Field object with a data type other than Text, the Type property setting automatically determines the Size property setting; you don't need to set it. For a Field object with the Text data type, however, you can set Size to any integer up to the maximum text size (255 for Microsoft Jet databases). If you do not set the size, the field will be as large as the database allows.

For Long Binary and Memo Field objects, Size is always set to 0. Use the FieldSize property of the Field object to determine the size of the data in a specific record. The maximum size of a Long Binary or Memo field is limited only by your system resources or the maximum size that the database allows.
Sort Property

Sets or returns the sort order for records in a Recordset object (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String that contains the ORDER BY clause of an SQL statement without the reserved words ORDER BY.

Remarks

You can use the Sort property with dynaset- and snapshot-type Recordset objects.

When you set this property for an object, sorting occurs when a subsequent Recordset object is created from that object. The Sort property setting overrides any sort order specified for a QueryDef object.

The default sort order is ascending (A to Z or 0 to 100).

The Sort property doesn't apply to table- or forward-only–type Recordset objects. To sort a table-type Recordset object, use the Index property.
Note In many cases, it's faster to open a new Recordset object by using an SQL statement that includes the sorting criteria.
Source Property

Returns the name of the object or application that originally generated the error.

Return Values

The return value is a String representing the object or application that generated the error.

Remarks

The Source property value is usually the object's class name or programmatic ID. Use the Source property to provide your users with information when your code is unable to handle an error generated in an object in another application.

For example, if you access Microsoft Excel and it generates a "Division by zero" error, Microsoft Excel sets Error.Number to the Microsoft Excel code for that error and sets the Source property to Excel.Application. Note that if the error is generated in another object called by Microsoft Excel, Microsoft Excel intercepts the error and still sets Error.Number to the Microsoft Excel code. However, the other Error object properties (including Source) will retain the values as set by the object that generated the error. The Source property always contains the name of the object that originally generated the error.
Based on all of the error documentation, you can write code that will handle the error appropriately. If your error handler fails, you can use the Error object information to describe the error to your user, using the Source property and the other Error properties to give the user information about which object originally caused the error, the description of the error, and so forth.

**Note**  The **On Error Resume Next** construct may be preferable to **On Error GoTo** when dealing with errors generated during access to other objects. Checking the Error object property after each interaction with an object removes ambiguity about which object your code was accessing when the error occurred. Thus, you can be sure which object placed the error code in Error.Number, as well as which object originally generated the error (Error.Source).
SourceField, SourceTable Properties

- **SourceField** returns a value that indicates the name of the field that is the original source of the data for a Field object.

- **SourceTable** returns a value that indicates the name of the table that is the original source of the data for a Field object.

**Return Values**

The return value is a String specifying the name of the field or table that is the source of data.

**Remarks**

For a Field object, use of the SourceField and SourceTable properties depends on the object that contains the Fields collection that the Field object is appended to, as shown in the following table.

<table>
<thead>
<tr>
<th>Object appended to</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Not supported</td>
</tr>
<tr>
<td>QueryDef</td>
<td>Read-only</td>
</tr>
</tbody>
</table>
Recordset Read-only
Relation Not supported
TableDef Read-only

These properties indicate the original field and table names associated with a Field object. For example, you could use these properties to determine the original source of the data in a query field whose name is unrelated to the name of the field in the underlying table.

Note The SourceTable property will not return a meaningful table name if used on a Field object in the Fields collection of a table-type Recordset object.
SourceTableName Property

Sets or returns a value that specifies the name of a linked table or the name of a base table (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String that specifies a table name. For a base table, the setting is a zero-length string ("""). This property setting is read-only for a base table and read/write for a linked table or an object not appended to a collection.
Sets or returns the SQL statement that defines the query executed by a QueryDef object.

**Settings and Return Values**

The setting or return value is a String that contains an SQL statement.

**Remarks**

The SQL property contains the SQL statement that determines how records are selected, grouped, and ordered when you execute the query. You can use the query to select records to include in a Recordset object. You can also define action queries to modify data without returning records.

The SQL syntax used in a query must conform to the SQL dialect of the query engine, which is determined by the type of workspace. In a Microsoft Jet workspace, use the Microsoft Jet SQL dialect, unless you create an SQL pass-through query, in which case you should use the dialect of the server. In an ODBCDirect workspace, use the SQL dialect of the server.

**Note**  You can send DAO queries to a variety of different database servers with
ODBCDirect, and different servers will recognize slightly different dialects of SQL. Therefore, context-sensitive Help is no longer provided for Microsoft Jet SQL, although online Help for Microsoft Jet SQL is still included through the Help menu. Be sure to check the appropriate reference documentation for the SQL dialect of your database server when using either ODBCDirect connections or pass-through queries in Microsoft Jet-connected client/server applications.

If the SQL statement includes **parameters** for the query, you must set these before execution. Until you reset the parameters, the same parameter values are applied each time you execute the query.

In an ODBCDirect workspace, you can also use the **SQL** property to execute a prepared statement on the server. For example, setting the **SQL** property to the following string will execute a prepared statement named “GetData” with one parameter on a Microsoft SQL Server back-end.

```
{call GetData (?)}
```

In a Microsoft Jet workspace, using a **QueryDef** object is the preferred way to perform SQL pass-through operations on Microsoft Jet-connected ODBC data sources. By setting the **QueryDef** object’s **Connect** property to an ODBC data source, you can use non–Microsoft-Jet-database SQL in the query to be passed to the external server. For example, you can use TRANSACT SQL statements (with Microsoft SQL Server or Sybase SQL Server databases), which the Microsoft Jet database engine would otherwise not process.

**Note** If you set the property to a string concatenated with a non-integer value, and the system parameters specify a non-U.S. decimal character such as a comma (for example, strSQL = "PRICE > " & lngPrice, and lngPrice = 125,50), an error will result when you try to execute the **QueryDef** object in a Microsoft Jet database. This is because during concatenation, the number will be converted to a string using your system's default decimal character, and Microsoft Jet SQL only accepts U.S. decimal characters.
StillExecuting Property

Indicates whether or not an asynchronous operation (that is, a method called with the `dbRunAsync` option) has finished executing (ODBCDirect workspaces only).

**Settings And Return Values**

The return value is a **Boolean** that is **True** if the query is still executing, and **False** if the query has completed.

**Remarks**

Use the **StillExecuting** property to determine if the most recently called asynchronous **Execute**, **MoveLast**, **OpenConnection**, or **OpenRecordset** method (that is, a method executed with the `dbRunAsync` option) is complete. While the **StillExecuting** property is **True**, any returned object cannot be accessed.

The following table shows what method is evaluated when you use **StillExecuting** on a particular type of object.

- **This asynchronous method is**
<table>
<thead>
<tr>
<th>If StillExecuting is used on</th>
<th>evaluated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection</td>
<td>Execute or OpenConnection</td>
</tr>
<tr>
<td>QueryDef</td>
<td>Execute</td>
</tr>
<tr>
<td>Recordset</td>
<td>MoveLast or OpenRecordset</td>
</tr>
</tbody>
</table>

Once the **StillExecuting** property on a [Connection](#) or [Recordset](#) object returns **False**, following the [OpenConnection](#) or [OpenRecordset](#) call that returns the associated [Recordset](#) or [Connection](#) object, the object can be referenced. So long as **StillExecuting** remains **True**, the object may not be referenced, other than to read the **StillExecuting** property. When you use the [NextRecordset](#) method to complete processing of a [Recordset](#), the **StillExecuting** property is reset to **True** while subsequent result sets are retrieved.

Use the [Cancel](#) method to terminate execution of a task in progress.
SystemDB Property

Sets or returns the path for the current location of the workgroup information file (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String describing the fully resolved path to the workgroup information file.

Remarks

The Microsoft Jet database engine allows you to define a workgroup and set different access permissions to each object in the database for each user in the workgroup. The workgroup is defined by the workgroup information file, typically called "system.mdw". For users to gain access to the secured objects in your database, DAO must have the location of this workgroup information file. The location can be identified to DAO either by specifying it in the Windows Registry or by setting the SystemDB property. On setup, the default setting is simply "system.mdw" with no path.

For this option to have any effect, you must set the SystemDB property before your application initializes the DBEngine object (that is, before creating an
instance of any other DAO object). The scope of this setting is limited to your application and can't be changed without restarting your application.
Table Property

Indicates the name of a Relation object's primary table. This should be equal to the Name property setting of a TableDef or QueryDef object (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a String that evaluates to the name of a table in the TableDefs collection or query in the QueryDefs collection. The Table property setting is read/write for a new Relation object not yet appended to a collection and read-only for an existing Relation object in a Relations collection.

Remarks

Use the Table property with the ForeignTable property to define a Relation object, which represents the relationship between fields in two tables or queries. Set the Table property to the Name property setting of the primary TableDef or QueryDef object, and set the ForeignTable property to the Name property setting of the foreign (referencing) TableDef or QueryDef object. The Attributes property determines the type of relationship between the two objects.

For example, if you had a list of valid part codes (in a field named PartNo)
stored in a ValidParts table, you could establish a one-to-many relationship with an OrderItem table such that if a part code were entered into the OrderItem table, it would have to already be in the ValidParts table. If the part code didn't exist in the ValidParts table and you had not set the Attributes property of the Relation object to dbRelationDontEnforce, a trappable error would occur.

In this case, the ValidParts table is the primary table, so the Table property of the Relation object would be set to ValidParts and the ForeignTable property of the Relation object would be set to OrderItem. The Name and ForeignName properties of the Field object in the Relation object's Fields collection would be set to PartNo.

The following illustration depicts this relation.
Transactions Property

Returns a value that indicates whether an object supports transactions.

**Return Values**

The return value is a **Boolean** data type that is **True** if the object supports transactions.

**Remarks**

In an ODBCDirect workspace, the **Transactions** property is available on Connection and Database objects, and indicates whether or not the ODBC driver you are using supports transactions.

In a **Microsoft Jet workspace**, you can also use the **Transactions** property with dynaset- or table-type **Recordset** objects. Snapshot- and forward-only–type **Recordset** objects always return **False**.

If a dynaset- or table-type **Recordset** is based on a **Microsoft Jet database engine** table, the **Transactions** property is **True** and you can use transactions. Other database engines may not support transactions. For example, you can't use transactions in a dynaset-type **Recordset** object based on a Paradox table.
Check the **Transactions** property before using the [BeginTrans](https://docs.microsoft.com/en-us/dotnet/api/system.data.recordset.begintrans) method on the **Recordset** object's [Workspace](https://docs.microsoft.com/en-us/dotnet/api/system.data.recordsetworkspace) object to make sure that transactions are supported. Using the **BeginTrans**, **CommitTrans**, or **Rollback** methods on an unsupported object has no effect.
**Type Property**

Sets or returns a value that indicates the operational type or data type of an object.

**Settings and Return Values**

The setting or return value is a constant that indicates an operational or data type. For a Field or Property object, this property is read/write until the object is appended to a collection or to another object, after which it's read-only. For a QueryDef, Recordset, or Workspace object, the property setting is read-only. For a Parameter object in a Microsoft Jet workspace the property is read-only, while in an ODBCDirect workspace the property is always read-write.

For a Field, Parameter, or Property object, the possible settings and return values are described in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbBigInt</td>
<td>Big Integer</td>
</tr>
<tr>
<td>dbBinary</td>
<td>Binary</td>
</tr>
<tr>
<td>dbBoolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>dbByte</td>
<td>Byte</td>
</tr>
</tbody>
</table>
For a `QueryDef` object, the possible settings and return values are shown in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Query type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbQAction</code></td>
<td>Action</td>
</tr>
<tr>
<td><code>dbQAppend</code></td>
<td>Append</td>
</tr>
<tr>
<td><code>dbQCompound</code></td>
<td>Compound</td>
</tr>
<tr>
<td><code>dbQCrosstab</code></td>
<td>Crosstab</td>
</tr>
<tr>
<td><code>dbQDDL</code></td>
<td>Data-definition</td>
</tr>
<tr>
<td><code>dbQDelete</code></td>
<td>Delete</td>
</tr>
<tr>
<td><code>dbQMakeTable</code></td>
<td>Make-table</td>
</tr>
<tr>
<td><code>dbQProcedure</code></td>
<td>Procedure (<code>ODBCDirect workspaces</code> only)</td>
</tr>
<tr>
<td><code>dbQSelect</code></td>
<td>Select</td>
</tr>
<tr>
<td><code>dbQSetOperation</code></td>
<td>Union</td>
</tr>
<tr>
<td></td>
<td>Used with <code>dbQSQLPassThrough</code> to specify a</td>
</tr>
</tbody>
</table>
**dbQSPTBulk** query that doesn't return records (Microsoft Jet workspaces only).

**dbQSQLPassThrough** Pass-through (Microsoft Jet workspaces only)

**dbQUpdate** Update

**Note** To create an SQL pass-through query in a Microsoft Jet workspace, you don't need to explicitly set the **Type** property to **dbQSQLPassThrough**. The Microsoft Jet database engine automatically sets this when you create a QueryDef object and set the **Connect** property.

For a **Recordset** object, the possible settings and return values are as follows.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Recordset type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbOpenTable</td>
<td>Table (Microsoft Jet workspaces only)</td>
</tr>
<tr>
<td>dbOpenDynamic</td>
<td>Dynamic (ODBCDirect workspaces only)</td>
</tr>
<tr>
<td>dbOpenDynaset</td>
<td>Dynaset</td>
</tr>
<tr>
<td>dbOpenSnapshot</td>
<td>Snapshot</td>
</tr>
<tr>
<td>dbOpenForwardOnly</td>
<td>Forward-only</td>
</tr>
</tbody>
</table>

For a **Workspace** object, the possible settings and return values are as follows.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Workspace type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbUseJet</td>
<td>The <strong>Workspace</strong> is connected to the Microsoft Jet database engine.</td>
</tr>
<tr>
<td>dbUseODBC</td>
<td>The <strong>Workspace</strong> is connected to an ODBC data source.</td>
</tr>
</tbody>
</table>

**Remarks**

When you append a new **Field**, **Parameter**, or **Property** object to the collection of an **Index**, **QueryDef**, **Recordset**, or **TableDef** object, an error occurs if the underlying database doesn't support the data type specified for the new object.
Unique Property

Sets or returns a value that indicates whether an Index object represents a unique (key) index for a table (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a Boolean that is True if the Index object represents a unique index. For an Index object, this property setting is read/write until the object is appended to a collection, after which it's read-only.

Remarks

A unique index consists of one or more fields that logically arrange all records in a table in a unique, predefined order. If the index consists of one field, values in that field must be unique for the entire table. If the index consists of more than one field, each field can contain duplicate values, but each combination of values from all the indexed fields must be unique.

If both the Unique and Primary properties of an Index object are set to True, the index is unique and primary: It uniquely identifies all records in the table in a predefined, logical order. If the Primary property is set to False, the index is a secondary index. Secondary indexes (both key and nonkey) logically arrange
records in a predefined order without serving as an identifier for records in the table.

Notes

- You don't have to create indexes for tables, but in large, unindexed tables, accessing a specific record can take a long time.

- Records retrieved from tables without indexes are returned in no particular sequence.

- The Attributes property of each Field object in the Index object determines the order of records and consequently determines the access techniques to use for that Index object.

- A unique index helps optimize finding records.

- Indexes don't affect the physical order of a base table indexes affect only how the records are accessed by the table-type Recordset object when a particular index is chosen or when the Microsoft Jet database engine creates Recordset objects.
Updatable Property

Returns a value that indicates whether you can change a DAO object.

Return Values

The return value is a Boolean data type that is True if the object can be changed or updated. (Snapshot- and forward-only–type Recordset objects always return False.)

Remarks

Depending on the object, if the Updatable property setting is True, the associated statement in the following table is true.

<table>
<thead>
<tr>
<th>Object</th>
<th>Type indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection</td>
<td>Data in the connected database can be changed.</td>
</tr>
<tr>
<td>Database</td>
<td>The object can be changed</td>
</tr>
<tr>
<td>QueryDef</td>
<td>The query definition can be changed</td>
</tr>
<tr>
<td>Recordset</td>
<td>The records can be updated</td>
</tr>
<tr>
<td>TableDef</td>
<td>The table definition can be changed</td>
</tr>
</tbody>
</table>
The **Updatable** property setting is always **True** for a newly created **TableDef** object and **False** for a linked **TableDef** object. A new **TableDef** object can be appended only to a database for which the current user has write permission.

Many types of objects can contain fields that can't be updated. For example, you can create a dynaset-type **Recordset** object in which only some fields can be changed. These fields can be fixed or contain data that increments automatically, or the dynaset can result from a query that combines updatable and nonupdatable tables.

If the object contains only read-only fields, the value of the **Updatable** property is **False**. When one or more fields are updatable, the property's value is **True**. You can edit only the updatable fields. A trappable error occurs if you try to assign a new value to a read-only field.

The **Updatable** property of a **QueryDef** object is set to **True** if the query definition can be updated, even if the resulting **Recordset** object isn't updatable.

Because an updatable object can contain read-only fields, check the **DataUpdatable** property of each field in the **Fields** collection of a **Recordset** object before you edit a record.
UpdateOptions Property

Sets or returns a value that indicates how the WHERE clause is constructed for each record during a batch update, and whether the batch update should use an UPDATE statement or a DELETE followed by an INSERT (ODBCCDirect workspaces only).

Settings And Return Values

The setting or return value is a Long that can be any of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbCriteriaKey</td>
<td>(Default) Uses just the key column(s) in the where clause.</td>
</tr>
<tr>
<td>dbCriteriaModValues</td>
<td>Uses the key column(s) and all updated columns in the where clause.</td>
</tr>
<tr>
<td>dbCriteriaAllCols</td>
<td>Uses the key column(s) and all the columns in the where clause.</td>
</tr>
</tbody>
</table>
| dbCriteriaTimeStamp       | Uses just the timestamp column if available
                            | (will generate a run-time error if no timestamp column is in the result set).|
**dbCriteriaDeleteInsert**
Uses a set of DELETE and INSERT statements for each modified row.

**dbCriteriaUpdate**
(Default) Uses an UPDATE statement for each modified row.

**Remarks**

When a batch-mode Update is executed, DAO and the client batch cursor library create a series of SQL UPDATE statements to make the needed changes. An SQL WHERE clause is created for each update to isolate the records that are marked as changed by the RecordStatus property. Because some remote servers use triggers or other ways to enforce referential integrity, it is often important to limit the fields being updated to just those affected by the change. To do this, set the UpdateOptions property to one of the constants `dbCriteriaKey`, `dbCriteriaModValues`, `dbCriteriaAllCols`, or `dbCriteriaTimeStamp`. This way, only the absolute minimum amount of trigger code is executed. As a result, the update operation is executed more quickly, and with fewer potential errors.

You can also concatenate either of the constants `dbCriteriaDeleteInsert` or `dbCriteriaUpdate` to determine whether to use a set of SQL DELETE and INSERT statements or an SQL UPDATE statement for each update when sending batched modifications back to the server. In the former case, two separate operations are required to update the record. In some cases, especially where the remote system implements DELETE, INSERT, and UPDATE triggers, choosing the correct UpdateOptions property setting can significantly impact performance.

If you don't specify any constants, `dbCriteriaUpdate` and `dbCriteriaKey` will be used.

Newly added records will always generate INSERT statements and deleted records will always generate DELETE statements, so this property only applies to how the cursor library updates modified records.
UserName Property

Sets or returns a value that represents a user, a group of users, or the owner of a Workspace object.

**Settings and Return Values**

The setting or return value is a String that evaluates to the name of a user. In a Microsoft Jet workspace, this represents a User object in the Users collection or a Group object in the Groups collection. For Microsoft Jet Container and Document objects, this property setting is read/write. For all Workspace objects, this property setting is read-only.

**Remarks**

Depending on the type of object, the UserName property represents the following.

- The owner of a Workspace object.

- A user or group of users when you manipulate the access permissions of a Container object or a Document object (Microsoft Jet workspaces only).
To find or set the permissions for a particular user or group of users, first set the **UserName** property to the user or group name that you want to examine. Then check the **Permissions** property setting to determine what permissions that user or group of users has, or set the **Permissions** property to change the permissions.

For a **Workspace** object, check the **UserName** property setting to determine the owner of the **Workspace** object. Set the **UserName** property to establish the owner of the **Workspace** object before you append the object to the **Workspaces** collection.
V1xNullBehavior Property

Indicates whether zero-length strings (""") used in code to fill Text or Memo fields are converted to Null.

Settings and Return Values

The setting or return value is a Boolean that is True if zero-length strings are converted to Null.

Remarks

This property applies to Microsoft Jet database engine version 1.x databases that have been converted to Microsoft Jet database engine version 2.0 or 3.0 databases.

Note The Microsoft Jet database engine automatically creates this property when it converts a version 1.x database to a version 2.0 or 3.x database. A 2.0 database will retain this property when it is converted to a 3.x database.

If you change this property setting, you must close and then reopen the database for your change to take effect.
For fastest performance, modify code that sets any Text or Memo fields to zero-length strings so that the fields are set to **null** instead, and remove the **V1xNullBehavior** property from the **Properties** collection.
ValidateOnSet Property

Sets or returns a value that specifies whether or not the value of a Field object is immediately validated when the object's Value property is set (Microsoft Jet workspaces only).

Settings and Return Values

The setting or return value is a Boolean that can be one of the following values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The validation rule specified by the ValidationRule property setting of the Field object is checked when you set the object's Value property.</td>
</tr>
<tr>
<td>False</td>
<td>(Default) Validate when the record is updated.</td>
</tr>
</tbody>
</table>

Only Field objects in Recordset objects support the ValidateOnSet property as read/write.

Remarks

Setting the ValidateOnSet property to True can be useful in a situation when a
user is entering records that include substantial Memo data. Waiting until the Update call to validate the data can result in unnecessary time spent writing the lengthy Memo data to the database if it turns out that the data was invalid anyway because a validation rule was broken in another field.
ValidationRule Property

Sets or returns a value that validates the data in a field as it's changed or added to a table (Microsoft Jet workspaces only).

Settings and Return Values

The settings or return values is a String that describes a comparison in the form of an SQL WHERE clause without the WHERE reserved word. For an object not yet appended to the Fields collection, this property is read/write. See Remarks for the more specific read/write characteristics of this property.

Remarks

The ValidationRule property determines whether or not a field contains valid data. If the data is not valid, a trappable run-time error occurs. The returned error message is the text of the ValidationText property, if specified, or the text of the expression specified by ValidationRule.

For a Field object, use of the ValidationRule property depends on the object that contains the Fields collection to which the Field object is appended.

Object appended to Usage
Index       Not supported
QueryDef    Read-only
Recordset   Read-only
Relation    Not supported
TableDef    Read/write

For a **Recordset** object, use of the **ValidationRule** property is read-only. For a **TableDef** object, use of the **ValidationRule** property depends on the status of the **TableDef** object, as the following table shows.

<table>
<thead>
<tr>
<th>TableDef</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base table</td>
<td>Read/write</td>
</tr>
<tr>
<td>Linked table</td>
<td>Read-only</td>
</tr>
</tbody>
</table>

Validation is supported only for databases that use the [Microsoft Jet database engine](https://msdn.microsoft.com/en-us/library/windows/desktop/ms676033%28v=vs.85%29.aspx).

The **string expression** specified by the **ValidationRule** property of a **Field** object can refer only to that **Field**. The expression can't refer to user-defined functions, SQL **aggregate functions**, or queries. To set a **Field** object's **ValidationRule** property when its **ValidateOnSet** property setting is **True**, the expression must successfully parse (with the field name as an implied operand) and evaluate to **True**. If its **ValidateOnSet** property setting is **False**, the **ValidationRule** property setting is ignored.

The **ValidationRule** property of a **Recordset** or **TableDef** object can refer to multiple fields in that object. The restrictions noted earlier in this topic for the **Field** object apply.

For a table-type **Recordset** object, the **ValidationRule** property inherits the **ValidationRule** property setting of the **TableDef** object that you use to create the table-type **Recordset** object.

For a **TableDef** object based on an **linked table**, the **ValidationRule** property inherits the **ValidationRule** property setting of the underlying **base table**. If the underlying base table doesn't support validation, the value of this property is a **zero-length string** ("").
**Note** If you set the property to a string concatenated with a non-integer value, and the system parameters specify a non-U.S. decimal character such as a comma (for example, `strRule = "PRICE > " & lngPrice, and `lngPrice = 125,50`), an error will result when your code attempts to validate any data. This is because during concatenation, the number will be converted to a string using your system's default decimal character, and Microsoft Jet SQL only accepts U.S. decimal characters.
**ValidationText Property**

Sets or returns a value that specifies the text of the **message** that your application displays if the value of a **Field** object doesn't satisfy the **validation rule** specified by the **ValidationRule** property setting (**Microsoft Jet workspaces** only).

**Settings and Return Values**

The setting or return value is a **String** that specifies the text displayed if a user tries to enter an invalid value for a field. For an object not yet appended to a collection, this property is read/write. For a **Recordset** object, this property setting is read-only. For a **TableDef** object, this property setting is read-only for a linked table and read/write for a base table.

**Remarks**

For a **Field** object, use of the **ValidationText** property depends on the object that contains the **Fields** collection to which the **Field** object is appended, as the following table shows.

<table>
<thead>
<tr>
<th>Object appended to</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Not supported</td>
</tr>
<tr>
<td>Type</td>
<td>Access</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>QueryDef</td>
<td>Read-only</td>
</tr>
<tr>
<td>Recordset</td>
<td>Read-only</td>
</tr>
<tr>
<td>Relation</td>
<td>Not supported</td>
</tr>
<tr>
<td>TableDef</td>
<td>Read/write</td>
</tr>
</tbody>
</table>
Value Property

Sets or returns the value of an object.

Settings and Return Values

The setting or return value is a Variant data type that evaluates to a value appropriate for the data type, as specified by the Type property of an object.

Remarks

Generally, the Value property is used to retrieve and alter data in Recordset objects.

The Value property is the default property of the Field, Parameter, and Property objects. Therefore, you can set or return the value of one of these objects by referring to them directly instead of specifying the Value property.

Trying to set or return the Value property in an inappropriate context (for example, the Value property of a Field object in the Fields collection of a TableDef object) will cause a trappable error.

Notes
In an ODBCDirect workspace, you cannot read or set the Value property of a Recordset field more than once without refreshing the current record. For example, to read and then set the Value property, first read the property, then use the Move 0 method to refresh the current record, then write the new value.

When reading decimal values from a Microsoft SQL Server database, they will be formatted using scientific notation through a Microsoft Jet workspace, but will appear as normal decimal values through an ODBCDirect workspace.
Version Property

- **Microsoft Jet workspace** — On the `DBEngine` object, returns the version of DAO currently in use. On the `Database` object, returns the version of Jet that created the .mdb file.

- **ODBCDirect workspace** — On the `DBEngine` object, returns the version of DAO currently in use. On the `Database` object, returns the version of the ODBC driver currently in use.

Return Values

The return value is a `String` that evaluates to a version number, formatted as follows.

- **Microsoft Jet workspace** represents the version number in the form "`major.minor`". For example, "3.0". The product version number consists of the version number (3), a period, and the release number (0).

- **ODBCDirect workspace** represents the DAO version number in the form "`major.minor`", or represents the ODBC driver version number in the form "`major.minor.build`". For example, the `DBEngine.Version` value of “3.5”
indicated DAO version 3.5. A **Database** object's **Version** value of 2.50.1032 indicates that the current instance of DAO is connected to ODBC version 2.5, build 1032.

**Remarks**

In a Microsoft Jet workspace, the **Version** property of a **Database** object corresponds to a version of the Microsoft Jet database engine, and doesn’t necessarily match the version number of the Microsoft product with which the database engine was included. For example, the **Version** property of a **Database** object created with Microsoft Visual Basic 3.0 will be 1.1, not 3.0.

The following table shows which version of the database engine was included with various versions of Microsoft products.

<table>
<thead>
<tr>
<th>Microsoft Jet Version (year released)</th>
<th>Microsoft Access</th>
<th>Microsoft Visual Basic</th>
<th>Microsoft Excel</th>
<th>Microsoft Visual C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 (1992)</td>
<td>1.0</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>1.1 (1993)</td>
<td>1.1</td>
<td>3.0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2.0 (1994)</td>
<td>2.0</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2.5 (1995)</td>
<td>N/A</td>
<td>4.0 (16-bit)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>3.0 (1995)</td>
<td>‘95 (7.0)</td>
<td>4.0 (32-bit)</td>
<td>‘95 (7.0)</td>
<td>4.x</td>
</tr>
<tr>
<td>3.5 (1996)</td>
<td>‘97 (8.0)</td>
<td>5.0</td>
<td>‘97 (8.0)</td>
<td>5.0</td>
</tr>
</tbody>
</table>
VisibleValue Property

Returns a value currently in the database that is newer than the OriginalValue property as determined by a batch update conflict (ODBCDirect workspaces only).

**Return Values**

The return value is a variant expression.

**Remarks**

This property contains the value of the field that is currently in the database on the server. During an optimistic batch update, a collision may occur where a second client modified the same field and record in between the time the first client retrieved the data and the first client's update attempt. When this happens, the value that the second client set will be accessible through this property.
**ODBCDirect**

A technology that allows you to access ODBC data sources directly by using DAO features that bypass the Microsoft Jet database engine.
**ODBC data source**

A term used to refer to a database or database server used as a source of data. ODBC data sources are referred to by their Data Source Name (DSN). Data sources can be created by using the Windows Control Panel or the `RegisterDatabase` method.
Microsoft Jet database engine

A database management system that retrieves data from and stores data in user and system databases. The Microsoft Jet database engine can be thought of as a data manager component with which other data access systems, such as Microsoft Access and Visual Basic, are built.
**Microsoft Jet database**

A database created with the Microsoft Jet database engine. The file name extension for a Microsoft Jet database is .mdb.
**ODBC (Open Database Connectivity)**

A standard protocol that permits applications to connect to a variety of external database servers or files. ODBC drivers used by the Microsoft Jet database engine permit access to Microsoft SQL Server and several other external databases.

The ODBC application programming interface (API) may also be used to access ODBC drivers and the databases they connect to without using the Microsoft Jet database engine.
**batch update**

A cursor model for clients that work with cursors but don't hold locks on the server or issue updates by row. Instead, the client updates many rows that are buffered locally, and then it issues a batch update. This cursor model also allows the client to drop the connection to the server and re-establish it with the same server or even a different server.

To use batch updating in DAO 3.5, you must use an ODBCDirect workspace, the `DefaultCursorDriver` property must be set to `dbUseClientBatchCursor` at the time the Connection is opened, and the `Recordset` must be opened with the `OpenRecordset` method's locktype argument set to `dbOptimisticBatch`. 
collision

A conflict that occurs during a batch update.

A collision occurs when a client reads data from the server and then attempts to modify that data in a batch update, but before the update attempt is actually executed another client changes the original server data. In this situation, the first client is attempting to modify server data without knowing what data actually exists on the server.
**replica**

A copy of a database, including its tables, queries, forms, reports, macros, and modules. A replica is a member of a replica set and can be synchronized with other replicas in the set. Changes to the data in a replicated table in one replica are sent and applied to the other members in the replica set.

**Note** The Design Master is also a replica.
**partial replica**

A database that contains only a subset of the records in a full replica. With a partial replica, you can set filters and identify relationships that define which subset of the records in the full replica should be present in the database.
Microsoft Jet workspace

A workspace that uses the Microsoft Jet database engine to access a data source. The data source can be a Microsoft Jet database file (.mdb), an ODBC database, such as a Paradox database, or an ISAM database.
**Microsoft Jet-connected ODBC data source**

An ODBC data source that is accessed by using Data Access Objects (DAO) and the Microsoft Jet database engine.
installable ISAM

A driver you can specify that allows access to external database formats such as dBASE, Microsoft Excel, and Paradox. ISAM is an acronym for Indexed Sequential Access Method. The Microsoft Jet database engine installs (loads) these ISAM drivers when referenced by your application. The location of these drivers is maintained in the Microsoft Windows Registry.
**ODBCDirect workspace**

A workspace that uses ODBCDirect to access an ODBC data source directly, bypassing the Microsoft Jet database engine.
asynchronous query

A type of query in which SQL queries return immediately, even though the results are still pending. This enables an application to continue with other processing while the query is pending completion.
**object variable**

A variable that contains a reference to an object.
DDL (Data Definition Language)

The language used to describe attributes of a database, especially tables, fields, indexes, and storage strategy.
**database replication**

The process of reproducing a database so that two or more copies (replicas) of the same database can stay synchronized. Changes to the data in a replicated table in one replica are sent and applied to the other replicas in the replica set. Changes made to the design of the database in the Design Master are sent and applied to all replicas in the set.
permission

One or more attributes that specify what kind of access a user has to data or objects in a database. For example, if a user has read data permission for a table or query, the user can view or retrieve but not edit data in the table or query.
method

A procedure similar to a function that operates on specific objects.
**property**

A named attribute of an object. Properties define object characteristics such as size, color, and screen location, or the state of an object, such as enabled or disabled.
replicate

Produce a replica, or copy, of something? for example, a database.
exclusive

A type of access that protects data in a database shared over a network. When you open a database in exclusive mode, you prevent others from opening the database.
one-to-one relationship

An association between two tables in which:

- The primary key value of each record in the primary table corresponds to the value in the matching field or fields of one and only one record in the related table.

- The primary key value of each record in the related table corresponds to the value in the matching field or fields of one and only one record in the primary table.
relationship

An association established between common fields (columns) in two tables. A relationship can be one-to-one, many-to-many, or one-to-many.
referential integrity

Rules that you set to establish and preserve relationships between tables when you add, change, or delete records. Enforcing referential integrity prohibits users from adding records to a joined table for which there is no primary key, changing values in a primary table that would result in orphaned records in a joined table, and deleting records from a primary table when there are matching related records.

If you select the **dbRelationDeleteCascade** or **dbRelationUpdateCascade** option for a relationship, the Microsoft Jet database engine allows changes and deletions but changes or deletes related records to make sure the rules are still enforced.
**linked table**

A table in another database linked to a Microsoft Jet database. Data for linked tables remains in the external database where it can be manipulated by other applications. (Formerly known as *attached tables*.)
**server-side cursor**

Cursors that reside on the server, as opposed to residing on the client computer. While client-side cursors copy the cursor to the workstation, server-side cursors use the resources of the database server to maintain the cursors.
**replicated database**

A database to which additional tables, fields, and properties have been added to record information about changes to data and the design of replicated objects in the database.
**action query**

A query that copies or changes data. Action queries include append, delete, make-table, and update queries. Delete and update queries change existing data; append and make-table queries copy existing data. In contrast, select queries return data records. An SQL pass-through query may also be an action query.
append query

An action query that adds new records to the end of an existing table or query. Append queries don't return records (rows).
compound query

A query that is composed of at least one action query (a query that copies or changes data) and at least one select query (a query that returns a Recordset without changing data). In DAO, a compound query is created by putting two or more SQL statements (separated by semicolons) in the SQL property of a QueryDef object.
**crosstab query**

A query that calculates a sum, average, count, or other type of total on records, and then groups the result by two types of information - one down the left side of a grid (row headings) and the other across the top (column headings). For example, the Quarterly Orders by Product query in the Northwind sample database is a crosstab query.
**data-definition query**

An SQL-specific query that can create, alter, or delete a table, or create or delete an index in a database.
**delete query**

An action query that deletes a set of rows that match the criteria you specify. A delete query doesn’t return rows.
**make-table query**

An action query that creates a new table from the `Recordset` object of an existing query.
**select query**

A query that asks a question about the data stored in your tables and returns a **Recordset** object without changing the data. Once the **Recordset** data is retrieved, you can examine and make changes to the data in the underlying tables. In contrast, action queries can make changes to your data, but they don't return data records.
pass-through query

An SQL-specific query you use to send commands directly to a SQL database server (such as Microsoft SQL Server). With pass-through queries, you work with the tables on the server instead of linking them. Pass-through queries are used to execute SQL queries and system-specific commands written by using SQL dialects known only to the server.

A pass-through query may or may not return records. If it does, they are always returned in a snapshot.
**update query**

An action query that changes a set of records according to criteria you specify. An update query doesn’t return any records.
inconsistent

The state of a multiple-table Recordset object that enables you to update all fields (columns). For example, in a Recordset created by joining two tables in a one-to-many join (as in a Customers and Orders table), you can update Orders.CustomerID so that it doesn’t match Customers.CustomerID, unless referential integrity disables the update.
consistent

The state of a multiple-table Recordset object that allows you to perform only updates that result in a consistent view of the data. For example, in a Recordset that is a join of two or more tables (a one-to-many relationship), a consistent query would not allow you to set the many-side key to a value that isn’t in the one-side table.
**ODBC Driver Manager**

An application that manages connections between ODBC-enabled data sources and the drivers used to access them.
connection string

A string used to define the source of data for an external database. The connection string is usually assigned to the Connect property of a QueryDef, TableDef, Connection, or Database object or as an argument to the OpenDatabase method.
A type of **Recordset** object that returns a dynamic set of pointers to live database data. Like a table- or a snapshot-type **Recordset**, a dynaset returns data in records (rows) and fields (columns). Unlike a table-type **Recordset**, a dynaset-type **Recordset** can be the result of a query that joins two or more tables. The records in a dynaset-type **Recordset** object are updatable if the **Updatable** property of the **Recordset** is **True**, the **Field** being changed is updatable, and the data page containing the current record isn’t locked. The data page is locked when the **Update** method is used (when the **LockEdits** property is **False**), or the **Edit** method is used (when the **LockEdits** property is **True**).
**SQL statement/string**

An expression that defines a Structured Query Language (SQL) command, such as SELECT, UPDATE, or DELETE, and may include clauses such as WHERE and ORDER BY. SQL strings and statements are typically used in queries, **Recordset** objects, and aggregate functions but can also be used to create or modify a database structure.
**DAO object**

An object that is defined by the Data Access Objects (DAO) library. You can use DAO objects, such as the **Database**, **TableDef**, **Recordset**, and **QueryDef** objects, to represent objects that are used to organize and manipulate data, such as tables and queries, in code.
group

A collection of user accounts in a **Workgroup** object, identified by group name and personal identifier (PID). Permissions assigned to a group apply to all users in the group.
**parameter query**

A query that requires you to provide one or more criteria values, such as Redmond for City, before the query is run. A parameter query isn’t, strictly speaking, a separate kind of query; rather, it extends the flexibility of other queries.
base table

A table in a Microsoft Jet database. A table defines the structure of a relational database, and is an object that stores data in records and fields. You can manipulate the structure of a base table by using the DAO objects or data definition SQL statements, and you can modify data in a base table by using Recordset objects or action queries.
**user account**

An account identified by a user name and personal identifier (PID) that is created to manage access to database objects in a Microsoft Jet database **Workgroup** object.
**session**

A session delineates a sequence of operations performed by the Microsoft Jet database engine. A session begins when a user logs on and ends when a user logs off. All operations performed during a session form one transaction scope and are subject to permissions determined by the logon user name and password. Sessions are implemented as **Workspace** objects by DAO.
persistent object

An object stored in the database; for example, a database table or QueryDef object. Dynaset-type or snapshot-type Recordset objects are not considered persistent objects because they are created in memory as needed.
transaction

A series of changes made to a database's data and schema. Mark the beginning of a transaction with the **BeginTrans** statement, commit the transaction by using the **CommitTrans** statement, and undo all your changes since **BeginTrans** by using the **Rollback** statement.

Transactions are optional and can be nested up to five levels. Transactions increase the speed of operations that change data and enable you to reverse changes easily.

Transactions are global to the referenced database object's **Workspace**.
Data Access Objects (DAO)

A programming interface to access and manipulate database objects.
**dynamic cursor**

A dynamic set of rows that you can use to add, change, or delete rows from an underlying database table or tables. A dynamic cursor can contain columns from one or more tables in a database. Membership is not fixed.
A dynaset-type **Recordset** object is a dynamic set of records that can contain fields from one or more tables or queries in a database and may be updatable. In an ODBCDirect database, a dynaset-type **Recordset** object corresponds to an ODBC **keyset cursor**.

**Remarks**

A dynaset-type **Recordset** object is a type of **Recordset** object you can use to manipulate data in an underlying database table or tables.

It differs from a snapshot-type **Recordset** object because the dynaset stores only the primary key for each record, instead of actual data. As a result, a dynaset is updated with changes made to the source data, while the snapshot is not. Like the table-type **Recordset** object, a dynaset retrieves the full record only when it's
needed for editing or display purposes.

To create a dynaset-type **Recordset** object, use the [OpenRecordset](#) method on an open database, against another dynaset- or snapshot-type **Recordset** object, on a [QueryDef](#) object, or on a [TableDef](#) object. (Opening **Recordset** objects on other **Recordset** objects or **TableDef** objects is available only in [Microsoft Jet workspaces](#).)

If you request a dynaset-type **Recordset** object and the [Microsoft Jet database engine](#) can't gain read/write access to the records, the Microsoft Jet database engine may create a read-only, dynaset-type **Recordset** object.

As users update data, the **base tables** reflects these changes. Therefore, current data is available to your application when you reposition the current record. In a [multiuser database](#), more than one user can open a dynaset-type **Recordset** object referring to the same records. Because a dynaset-type **Recordset** object is dynamic, when one user changes a record, other users have immediate access to the changed data. However, if one user adds a record, other users won’t see the new record until they use the **Requery** method on the **Recordset** object. If a user deletes a record, other users are notified when they try to access it.

Records added to the database don't become a part of your dynaset-type **Recordset** object unless you add them by using the [AddNew](#) and [Update](#) methods. For example, if you use an **action query** containing an INSERT INTO **SQL statement** to add records, the new records aren't included in your dynaset-type **Recordset** object until you either use the **Requery** method or you rebuild your **Recordset** object using the [OpenRecordset](#) method.

To maintain data integrity, the Microsoft Jet database engine can **lock** dynaset- and table-type **Recordset** objects during **Edit** (pessimistic locking) or **Update** operations (optimistic locking) so that only one user can update a particular record at a time. When the Microsoft Jet database engine locks a record, it locks the entire 2K **page** containing the record.

You can also use optimistic and pessimistic locking with non-ODBC tables. When you access external tables using [ODBC](#) through a Microsoft Jet workspace, you should always use optimistic locking. The **LockEdits** property and the **lockedits** parameter of the [OpenRecordset](#) method determine the locking conditions during editing.
Not all fields can be updated in all dynaset-type **Recordset** objects. To determine whether you can update a particular field, check the **DataUpdatable** property setting of the **Field** object.

A dynaset-type **Recordset** object may not be updatable if:

- There isn't a unique **index** on the ODBC or Paradox table or tables.
- The data page is locked by another user.
- The record has changed since you last read it.
- The user doesn't have **permission**.
- One or more of the tables or fields are read-only.
- The database is opened as read-only.
- The **Recordset** object was either created from multiple tables without a JOIN statement or the query was too complex.

The order of a dynaset-type **Recordset** object or **Recordset** data doesn't necessarily follow any specific sequence. If you need to order your data, use an SQL statement with an ORDER BY clause to create the **Recordset** object. You can also use a WHERE clause to filter the records so that only certain records are added to the **Recordset** object. Using SQL statements in this way to select a subset of records and order them usually results in faster access to your data than using the **Filter** and **Sort** properties.
Long data type

A fundamental data type that holds long-integer numbers. A Long variable is stored as a 32-bit (4-byte) number ranging in value from -2,147,483,648 to 2,147,483,647.
HelpContext, HelpFile Properties

- **HelpContext** — returns a context ID, as a `Long` variable, for a topic in a Help file.

- **HelpFile** — returns a `String` that is a fully qualified path to the Help file.

**Remarks**

If you specify a Help file in `HelpFile`, you can use the `HelpContext` property to automatically display the Help topic it identifies.

**Note**  You should write procedures in your application to handle typical errors. When programming with an object, you can use the Help supplied by the object's Help file to improve the quality of your error handling, or to display a meaningful message to your user if the error is not recoverable.
**current record**

The record in a **Recordset** object that you can use to modify or examine data. Use the Move methods to reposition the current record in a recordset. Use the Find methods (with a dynaset- or snapshot-type **Recordset** object) or the **Seek** method (with a table-type **Recordset** object) to change the current record position according to specific criteria.

Only one record in a **Recordset** can be the current record; however, a **Recordset** may have no current record. For example, after a dynaset-type **Recordset** record has been deleted, or when a **Recordset** has no records, the current record is undefined. In this case, operations that refer to the current record result in a trappable error.
**data type**

The characteristics of a variable that determine what kind of data the variable can hold. Data types include **Big Integer**, **Binary**, **Byte**, **Boolean**, **Char**, **Currency**, **Date**, **Decimal**, **Double**, **Float**, **GUID**, **Integer**, **Long**, **Long Binary (OLE Object)**, **Memo**, **Numeric**, **Single**, **String**, **Text**, **Time**, **TimeStamp**, **VarBinary**, **Variant** (default), user-defined types (created with the **Type** statement), and object data types, which include host-defined object data types and DAO object types.
**OLE Object data type**

A field data type you use for objects created in other applications that can be linked or embedded in a Microsoft Jet database. For example, you could use an OLE Object field to store a collection of pictures.
Memo data type

A field data type. Such fields can contain up to 1.2 GB of text data.
forward-only cursor

A result set where the current position can only move forward by a specified number of records, or to the very last record. The current position cannot be moved back to the beginning of the result set or to previous records. The cursor membership, order, and values are generally fixed when the cursor is opened. If other users update, delete, or insert rows, the cursor doesn't reflect these changes until you close and reopen it.
secure workgroup

A Microsoft Jet database **Workgroup** object to which users log on with a user name and password and in which access to database objects is restricted according to permissions specified for user accounts and groups.
Null

A value that indicates missing or unknown data. Null values can be entered in fields for which information is unknown and in expressions and queries. In Visual Basic, the Null keyword indicates a Null value. Some fields, such as those defined as containing the primary key, can’t contain Null values.
join

A database operation that combines some or all records from two or more tables, such as an equi-join, outer join, or self-join. Generally, a join refers to an association between a field in one table and a field of the same data type in another table. You create a join with an SQL statement.

When you define a relationship between two tables, you create a join by specifying the primary and foreign table fields. When you add a table to a query, you need to create a join between appropriate fields in the SQL statement that defines the query.
**primary key**

One or more fields whose value or values uniquely identify each record in a table. In a relationship, a primary key is used to refer to specific records in one table from another table. A primary key is called a foreign key when it is referred to from another table. You can have only one primary key. An Employees table, for example, could use the social security number for the primary key.
foreign key

One or more table fields that refer to the primary key field or fields in another table. A foreign key indicates how the tables are related – the data in the foreign key and primary key fields must match. For example, a list of valid part numbers would contain a foreign key to an inventory table containing references to valid part numbers. Used when establishing referential integrity for a database.
**cascading update**

For relationships enforcing referential integrity between tables, an option that causes a change to the primary key in a record in the primary table to automatically update the foreign key in all related records in the related foreign table or tables.

For example, suppose you establish a relationship between a Customers (primary) table and an Orders (foreign) table with the cascading update option enabled. When the primary key in a record in the Customers table is changed, all orders associated with that customer would also be changed in the foreign Orders table.
cascading delete

For relationships that enforce referential integrity between tables, an option that causes the deletion of a record from the primary table to automatically delete all related records in the related foreign table or tables.

For example, suppose you establish a relationship between a Customers (primary) table and an Orders (foreign) table with the cascading delete option enabled. When a record in the Customers table is deleted, all orders associated with that customer would also be deleted in the foreign Orders table.
**String data type**

A fundamental data type that holds character information. A **String** variable is either fixed-length or variable-length and contains one character per byte. Fixed-length strings are declared to be a specific length and can contain 1 to approximately 64K (2^16) characters. Variable-length strings can be any length up to 2 billion (2^31) characters (approximately 64K [2^16] characters for Microsoft Windows version 3.1 and earlier), less a small amount of storage overhead.
**Integer data type**

A fundamental data type that holds integer numbers. An **Integer** variable is stored as a 16-bit (2-byte) number ranging in value from -32,768 to 32,767.
Variant data type

A special data type that can contain numeric, string, or date data as well as the special values **Empty** and **Null**. The **VarType** function defines how the data in a **Variant** is treated. All variables become variant types if not explicitly declared as some other type.
Boolean data type

A True/False or yes/no value. **Boolean** values are usually stored in Bit fields in a Microsoft Jet database; however, some databases don't support this data type directly.
**zero-length string**

A string containing no characters ("""). The **Len** function of a zero-length string returns 0.
**query**

A formalized instruction to a database to either return a set of records or perform a specified action on a set of records as specified in the query. For example, the following SQL query statement returns records:

```
SELECT CompanyName FROM Publishers WHERE Region = 'NY'
```

You can create and run select, action, crosstab, parameter, and SQL-specific queries.
**record**

A set of related data about a person, place, event, or some other item. Table data is stored in records (rows) in the database. Each record is composed of a set of related fields (columns)—each field defining one attribute of information for the record. Taken together, a record defines one specific unit of retrievable information in a database.
**field**

A category of information stored in a table in a database—a column of data. An element of a database table that contains a specific item of information, such as last name.
**keyset cursor**

A set of rows that you can use to add, change, or delete rows from an underlying database table or tables. Movement within the keyset is unrestricted. A keyset cursor can contain columns from one or more tables in a database. Membership is fixed.
**static cursor**

A result set where the membership, order, and values are generally fixed when the cursor is opened. If other users update, delete, or insert rows, the cursor doesn't reflect these changes until you close and reopen it.
**scope**

The attribute of a variable or procedure that determines which sections of which modules recognize it. There are three levels of scope: public, module, and procedure. Variables that you declare with **Public** can be accessed by any module, while variables that you declare in a specific module can be used only within that module. Also, variables that you declare in a **Sub** or **Function** procedure can be used only in that particular procedure.
A TEMP environment variable is set by your autoexec.bat file when you start your system. Generally, TEMP points to an area on your hard disk used by Microsoft Windows and other programs, like the Microsoft Jet database engine, to store information that doesn't need to be saved after you shut down your system. For example, the following line in your autoexec.bat file points the TEMP environment variable to the D:\TempArea folder:

SET TEMP=D:\TempArea
primary table

The "one" side of two related tables in a one-to-many relationship with a foreign table. Generally, a primary key table is used to establish or enforce referential integrity.
foreign table

A table that provides a foreign key to another table in the database. Generally, you use a foreign table to establish or enforce referential integrity. The foreign table is usually on the "many" side of a one-to-many relationship. An example of a foreign table is a table of customer orders.
**one-to-many relationship**

An association between two tables in which:

- The primary key value of each record in the primary table corresponds to the value in the matching field or fields of many records in the related table.

- The primary key value of each record in the related table corresponds to the value in the matching field or fields of one and only one record in the primary table.
left join

A left outer join includes all of the records from the first (left) of two tables, even if there are no matching values for records in the second (right) table.
right join

A right outer join includes all of the records from the second (right) of two tables, even if there are no matching values for records in the first (left) table.

For example, you could use LEFT JOIN with the Departments (left) and Employees (right) tables to select all departments, including those that have no employees assigned to them. To select all employees, including those who aren't assigned to a department, you would use RIGHT JOIN.
index

A dynamic cross-reference of one or more table data fields (columns) that permits faster retrieval of specific records from a table. As records are added, changed, or deleted, the database management system automatically updates the index to reflect the changes.

When used with a table-type Recordset object, the current index determines the order in which data records are returned to the Recordset. A table may have several indexes defined for its data.

A DAO Index object represents an index for a TableDef object.
locked

The condition of a data page, **Recordset** object, or **Database** object that makes it read-only to all users except the one who is currently entering data in it.
A portion of the database in which record data is stored. Depending on the size of the records, a page may contain more than one record. In Microsoft Jet databases (.mdb), a page is 2048 (2K) bytes in length.
pessimistic

A type of locking in which the page containing one or more records, including the record being edited, is unavailable to other users when you use the Edit method, and remains unavailable until you use the Update method. Pessimistic locking is enabled when the LockEdits property of the Recordset object is set to True.
**optimistic**

A type of locking in which the data page containing one or more records, including the record being edited, is unavailable to other users only while the record is being updated by the **Update** method, but is available between the **Edit** and **Update** methods. Optimistic locking is used when accessing ODBC databases or when the **LockEdits** property of the **Recordset** object is set to **False**.
**default workspace**

The *Workspace* object that DAO automatically establishes when your application first references any DAO object. This *Workspace* is referenced by `DBEngine.Workspaces(0)` or simply `Workspaces(0)`. 
database engine

That part of the database system that retrieves data from and stores data in user and system databases.
sort order

A sequencing principle used to order data, alphabetically or numerically. The sort order can be either ascending or descending.
**bookmark**

A property of the *Recordset* object that contains a binary string identifying the current record. If you assign the *Bookmark* value to a variable and then move to another record, you can make the earlier record current again by setting the *Bookmark* property to that string variable.
**collection**

An object that contains a set of related objects. An object's position in the collection can change whenever a change occurs in the collection; therefore, the position of any specific object in the collection may vary.
string expression

Any expression that evaluates to a sequence of contiguous characters. Elements of the expression can include a function that returns a string, a string literal, a string constant, a string variable, a string Variant, or a function that returns a string Variant (VarType 8).
**Long Binary data type**

A type of field that can hold 1.2 GB of data. Long Binary fields can contain any type of binary data. (Also known as an *OLE Object* data type in Microsoft Access).
current transaction

All changes made to a Recordset object after you use the last BeginTrans method and before you use the Rollback or CommitTrans method.
update

The process that saves changes to data in a record. Until the record is saved, changes are stored in a temporary record called the copy buffer. The UPDATE clause in an SQL statement changes data values in one or more records (rows) in a database table.
forward-only – type Recordset

A Recordset object in which records can be searched only from beginning to end; the current record position can't be moved back to the first record. Forward-only – type recordsets reduce processing overhead on remote databases. For example, you can use a forward-only – type Recordset on a linked remote table to quickly process data in one pass, such as when you’re building a customized report.
**personal identifier (PID)**

A case-sensitive alphanumeric string 4-20 characters long that the Microsoft Jet database engine uses in combination with the account name to identify a user or group in a **Workgroup** object. You provide the PID and the account name when you create a new user or group.
replicated object

A table, query, form, report, macro, or module that is in all replicas in the replica set. You can change the replicated object only at the Design Master, and these changes are dispersed to other replicas in the replica set during a synchronization.
**local object**

A table, query, form, report, macro, or module that remains in the replica where it was created. Neither the object nor changes to the object are dispersed to other members in the replica set.
ASCII Character Set

American Standard Code for Information Interchange (ASCII) 7-bit character set widely used to represent letters and symbols found on a standard U.S. keyboard. The ASCII character set is the same as the first 128 characters (0 – 127) in the ANSI character set.
**copy buffer**

A location created by the Microsoft Jet database engine for the contents of a record that is open for editing. The **Edit** method copies the current record to the copy buffer; the **AddNew** method clears the buffer for a new record and sets the default values; and the **Update** method saves the data from the copy buffer to the database, replacing the current record or inserting the new record. Any statement that resets or moves the current record pointer discards the copy buffer. For example, using the **MoveNext** method or changing the **Index** property of a table would discard the contents of the copy buffer.
array

A variable that contains a finite number of elements that have a common name and data type. Each element of an array is identified by a unique index number. Changes made to one element of an array do not affect the other elements.
run-time error

An error that occurs when code is running. A run-time error results when a statement attempts an invalid operation.
replica set

Replicas that share the same database design and unique replica set identifier. Synchronization occurs between replica set members.
current index

For an indexed table-type **Recordset** object, the index most recently set with the **Index** property. This index is the basis for ordering records in a table-type **Recordset**, and is used by the **Seek** method to locate records. A **Recordset** object can have more than one index but can use only one index at a time (although a **TableDef** object may have several indexes defined on it). The Microsoft Jet database engine may use more than one index to evaluate a query.
**case-sensitive**

Capable of distinguishing between uppercase and lowercase letters. A case-sensitive search finds only text that is an exact match of uppercase and lowercase letters. Such a search would, for instance, treat “ZeroLengthStr” and “zerolengthstr” as different. Microsoft Jet database operations are not case-sensitive. However, case sensitivity is a feature of some other database management systems.
**ODBC driver**

A dynamic-link library (DLL) used to connect a specific Open Database Connectivity data source with another (client) application.
dll (dynamic-link library)

A set of routines that can be called from procedures and are loaded and linked into your application at run time.
field data types

The following table lists the Field data types.

<table>
<thead>
<tr>
<th>DAO Field data type</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Integer</td>
<td>dbBigInt</td>
</tr>
<tr>
<td>Binary</td>
<td>dbBinary</td>
</tr>
<tr>
<td>Boolean</td>
<td>dbBoolean</td>
</tr>
<tr>
<td>Byte</td>
<td>dbByte</td>
</tr>
<tr>
<td>Char</td>
<td>dbChar</td>
</tr>
<tr>
<td>Currency</td>
<td>dbCurrency</td>
</tr>
<tr>
<td>Date/Time</td>
<td>dbDate</td>
</tr>
<tr>
<td>Decimal</td>
<td>dbDecimal</td>
</tr>
<tr>
<td>Double</td>
<td>dbDouble</td>
</tr>
<tr>
<td>Float</td>
<td>dbFloat</td>
</tr>
<tr>
<td>GUID</td>
<td>dbGUID</td>
</tr>
<tr>
<td>Integer</td>
<td>dbInteger</td>
</tr>
<tr>
<td>Long</td>
<td>dbLong</td>
</tr>
<tr>
<td>Long Binary (OLE Object)</td>
<td>dbLongBinary</td>
</tr>
<tr>
<td>Memo</td>
<td>dbMemo</td>
</tr>
<tr>
<td>Numeric</td>
<td>dbNumeric</td>
</tr>
<tr>
<td>Single</td>
<td>dbSingle</td>
</tr>
<tr>
<td>Text</td>
<td>dbText</td>
</tr>
<tr>
<td>Time</td>
<td>dbTime</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>dbTimeStamp</td>
</tr>
<tr>
<td>VarBinary</td>
<td>dbVarBinary</td>
</tr>
</tbody>
</table>
Internet

A worldwide network of thousands of smaller computer networks and millions of commercial, educational, government, and personal computers. The Internet is like an electronic city with virtual libraries, storefronts, business offices, art galleries, and so on.
**Design Master**

A database to which system tables, system fields, and replication properties have been added. A Design Master is the first replica in a replica set. You can make changes to the database structure only with the Design Master. Replicas in the same replica set can take turns being the Design Master, but there can be only one Design Master at a time in each replica set.
**synchronization**

The process of updating two replicas in which all updated records and objects are exchanged. The exchange of data between two replicas can be one-way or two-way and may be handled by a Synchronizer.
**Text data type**

A field data type. Text fields can contain up to 255 characters or the number of characters specified by the `Size` property of the `Field` object, whichever is less. If the `Size` property of the text field is set to 0, the text field can hold up to 255 characters of data.
current database

The **Database** object returned by the **CurrentDB( )** function. A reference of `DBEngine.Workspaces(0).Databases(0)` returns the first database opened. This concept applies only to Microsoft Access.
variant expression

Any expression that can evaluate to numeric, string, or date data as well as the special values Empty and Null.
Byte data type

A fundamental data type used to hold small positive integer numbers ranging from 0 to 255.
**ODBCDirect data source**

A data source that is accessed by using the DAO ODBCDirect features, which bypass the Microsoft Jet database engine.
clustered index

The physical order of rows is the same as the indexed order of rows.
**string comparison**

The use of an operator to determine whether one string is greater than or equal to another string. If you use **Option Compare Text** in the Declarations section of a module, string comparisons are not case-sensitive. If you use **Option Compare Binary**, comparisons are case-sensitive. If you use **Option Compare Database**, the comparison method is set by the current database.
conflict table

A table that is generated when a synchronization conflict occurs between two replicas. The conflict table appears only in the replica with rejected changes.
**object expression**

An expression that specifies a particular object. This expression can include any of the object's containers. For example, if your application has an Application object that contains a Document object that contains a Text object, the following are valid object expressions:

```
Application.Document.Text
Application.Text
Document.Text
Text
```
expression

Any combination of operators, constants, literal values, functions, and names of fields, controls, and properties that evaluates to a single value. You can use expressions as settings for many properties and action arguments, to set criteria, or define calculated fields in queries.
aggregate function

A function, such as **Sum, Count, Avg, and Var**, that you can use to calculate totals. In writing expressions and in programming, you can use SQL aggregate functions (including the four listed here) and domain aggregate functions to determine various statistics.
**AutoNumber field**

A field data type that automatically stores a unique number for each record as it's added to a table. An AutoNumber field always uses the **Long** data type, and numbers generated by an AutoNumber field can't be modified. (Also known as a Counter field.)
**GUID data type**

Globally Unique Identifier/Universally Unique Identifier. A unique identification string used with remote procedure calls. Every interface and object class uses a **GUID** for identification. A **GUID** is a 128-bit value. For example, 12345678-1234-1234-1234-123456789ABC is a syntactically correct **GUID**. The **GUIDs** on the client and server must match for the client and server to bind. Vendors of objects can request that Microsoft allocate one or more sets of 256 **GUIDs** for their exclusive use. Alternatively, if you have a network card, you can run a tool named Uuidgen.exe, which provides a set of 256 **GUIDs** based on the time of day, the date, and a unique number contained in your network card.
filter

A set of criteria applied to records in order to create a subset of the records.
**instance**

Any one of a set of objects sharing the same class. For example, multiple instances of a **Form** class share the same code and are loaded with the same controls with which the **Form** class was designed. During run time, the individual properties of controls on each instance can be set to different values.
Single data type

A fundamental data type that holds single-precision floating-point numbers in IEEE format. A **Single** variable is stored as a 32-bit (4-byte) number ranging in value from -3.402823E38 to -1.401298E-45 for negative values, from 1.401298E-45 to 3.402823E38 for positive values, and 0.
class

The formal definition of an object. The class acts as the template from which an instance of an object is created at run time. The class defines the properties of the object and the methods used to control the object's behavior.
**parameter**

An element containing a value that you can change to affect the results of the query. For example, a query returning data about an employee might have a parameter for the employee's name. You can then use one `QueryDef` object to find data about any employee by setting the parameter to a specific name before running the query.
Boolean expression

An expression that evaluates to either True or False.
**Big Integer data type**

A data type that stores a signed, exact numeric value with precision 19 (signed) or 20 (unsigned), scale 0 (signed: \(-263 \leq n \leq 263-1\); unsigned: \(0 \leq n \leq 264-1\)).
**Binary data type**

A data type that stores fixed-length binary data. The maximum length is 255 bytes.
Char data type

A data type that stores a fixed-length character string. The length is set by the Size property.
**Currency data type**

A data type that is useful for calculations involving money or for fixed-point calculations in which accuracy is extremely important. This data type is used to store numbers with up to 15 digits to the left of the decimal point and 4 digits to the right. Because the Currency data type uses discrete values for all amounts, binary round-off isn’t a factor when calculating totals.
dates and times

Dates and times are stored internally as different parts of a real number.

The value to the left of the decimal represents a date between December 30, 1899 and December 30, 9999, inclusive. Negative values represent dates prior to December 30, 1899.

The value to the right of the decimal represents a time between 0:00:00 and 23:59:59, inclusive. Midday is represented by .5.
**Decimal data type**

A data type that stores a signed, exact numeric value with precision $p$ and scale $s$ ($1 \leq p \leq 15; 0 \leq s \leq p$).
**Double data type**

A fundamental data type that holds double-precision floating-point numbers in IEEE format. A **Double** variable is stored as a 64-bit (8-byte) number ranging in value from $-1.79769313486231E308$ to $-4.94065645841247E-324$ for negative values, from $4.94065645841247E-324$ to $1.79769313486231E308$ for positive values, and 0.
Float data type

A data type that stores a signed, approximate numeric value with mantissa precision 15 (zero or absolute value 10^-308 to 10^308).
**Numeric data type**

A data type that stores a signed, exact numeric value with precision $p$ and scale $s$ ($1 \leq p \leq 15$; $0 \leq s \leq p$).
**Time data type**

A data type that stores a time value. The value is dependent on the clock setting of the data source.
**TimeStamp data type**

A data type that stores a TimeStamp. The value is dependent on the clock setting of the data source.
VarBinary data type

A data type that stores variable-length binary data. The maximum length is 255 bytes.
procedural query

An SQL statement that executes a stored procedure.
**union query**

An SQL-specific select query that creates a snapshot-type *Recordset* object containing data from all specified records in two or more tables with any duplicate records removed. To include the duplicates, add the keyword ALL.

For instance, a union query of the Customers table and the Suppliers table results in a snapshot-type *Recordset* that contains all suppliers that are also customers.
client batch cursor library

A library that provides client-side cursor support for ODBCDirect database applications. This library supports all four types of cursors (keyset, static, dynamic, and forward-only) and provides a number of other features including the ability to dissociate connections and perform optimistic batch updates.
validation

The process of checking whether entered data meets certain conditions or limitations.
validation rule

A rule that sets limits or conditions on what can be entered in one or more fields. Validation rules can be set for a Field or TableDef object. Validation rules are checked when you update a record containing fields requiring validation. If the rule is violated, a trappable error results.
**message**

A packet of information passed from one application to another.
**multiuser database**

A database that permits more than one user to access and modify the same set of data at the same time. In some cases, the additional "user" may be another instance of your application or another application running on your system that accesses the same data as some other application.