Recordset (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

A CRecordset object represents a set of records selected from a data source. The records can be from:

- A table.
- A query.
- A stored procedure that accesses one or more tables.

An example of a recordset based on a table is "all customers," which accesses a Customer table. An example of a query is "all invoices for Joe Smith." An example of a recordset based on a stored procedure (sometimes called a predefined query) is "all of the delinquent accounts," which invokes a stored procedure in the back-end database. A recordset can join two or more tables from the same data source, but not tables from different data sources.

Note

Some ODBC drivers support views of the database. A view in this sense is a query originally created with the SQL `CREATE VIEW` statement.

Recordset Capabilities

All recordset objects share the following capabilities:

- If the data source is not read-only, you can specify that your recordset be updatable, appendable, or read-only. If the recordset is updateable, you can choose either pessimistic or optimistic locking methods, provided the driver supplies the appropriate locking support. If the data source is read-only, the recordset will be read-only.
- You can call member functions to scroll through the selected records.

- You can filter the records to constrain which records are selected from those available.
- You can sort the records in ascending or descending order, based on one or more columns.
- You can parameterize the recordset to qualify the recordset selection at run time.

Snapshots and Dynasets

There are two principal types of recordsets: snapshots and dynasets. Both are supported by class `CRecordset`. Each shares the common characteristics of all recordsets, but each also extends the common functionality in its own specialized way. Snapshots provide a static view of the data and are useful for reports and other situations in which you want a view of the data as it existed at a particular time. Dynasets are useful when you want updates made by other users to be visible in the recordset without having to requery or refresh the recordset. Snapshots and dynasets can be updateable or read-only. To reflect records added or deleted by other users, call CRecordset::Requery.

`CRecordset` also allows for two other types of recordsets: dynamic recordsets and forward-only recordsets. Dynamic recordsets are similar to dynasets; however, dynamic recordsets reflect any records added or deleted without calling `CRecordset::Requery`. For this reason, dynamic recordsets are generally expensive with respect to processing time on the DBMS, and many ODBC drivers do not support them. In contrast, forward-only recordsets provide the most efficient method of data access for recordsets that do not require updates or backward scrolling. For example, you might use a forward-only recordset to migrate data from one data source to another, where you only need to move through the data in a forward direction. To use a forward-only recordset, you must do both of the following:

- Pass the option `CRecordset::forwardOnly` as the *nOpenType* parameter of the Open member function.
- Specify `CRecordset::readOnly` in the *dwOptions* parameter of `Open`.

> **Note**
>
> For information about ODBC driver requirements for dynaset support, see

ODBC. For a list of ODBC drivers included in this version of Visual C++ and for information about obtaining additional drivers, see **ODBC Driver List**.

Your Recordsets

For every distinct table, view, or stored procedure that you want to access, you typically define a class derived from `CRecordset`. (The exception is a database join, in which one recordset represents columns from two or more tables.) When you derive a recordset class, you enable the record field exchange (RFX) mechanism or the bulk record field exchange (Bulk RFX) mechanism, which are similar to the dialog data exchange (DDX) mechanism. RFX and Bulk RFX simplify the transfer of data from the data source into your recordset; RFX additionally transfers data from your recordset to the data source. For more information, see Record Field Exchange (RFX) and Recordset: Fetching Records in Bulk (ODBC).

A recordset object gives you access to all the selected records. You scroll through the multiple selected records using `CRecordset` member functions, such as `MoveNext` and `MovePrev`. At the same time, a recordset object represents only one of the selected records, the current record. You can examine the fields of the current record by declaring recordset class member variables that correspond to columns of the table or of the records that result from the database query. For information about recordset data members, see Recordset: Architecture (ODBC).

The following topics explain the details of using recordset objects. The topics are listed in functional categories and a natural browse order to permit sequential reading.

Topics about the mechanics of opening, reading, and closing recordsets

- Recordset: Architecture (ODBC)
- Recordset: Declaring a Class for a Table (ODBC)
- Recordset: Creating and Closing Recordsets (ODBC)
- Recordset: Scrolling (ODBC)
- Recordset: Bookmarks and Absolute Positions (ODBC)

- Recordset: Filtering Records (ODBC)
- Recordset: Sorting Records (ODBC)
- Recordset: Parameterizing a Recordset (ODBC)

Topics about the mechanics of modifying recordsets

- Recordset: Adding, Updating, and Deleting Records (ODBC)
- Recordset: Locking Records (ODBC)
- Recordset: Requerying a Recordset (ODBC)

Topics about somewhat more advanced techniques

- Recordset: Performing a Join (ODBC)
- Recordset: Declaring a Class for a Predefined Query (ODBC)
- Recordset: Dynamically Binding Data Columns (ODBC)
- Recordset: Fetching Records in Bulk (ODBC)
- Recordset: Working with Large Data Items (ODBC)
- Recordset: Obtaining SUMs and Other Aggregate Results (ODBC)

Topics about how recordsets work

- Recordset: How Recordsets Select Records (ODBC)
- Recordset: How Recordsets Update Records (ODBC)

See also

Open Database Connectivity (ODBC)
MFC ODBC Consume
Transaction (ODBC)

Recordset: Architecture (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

This topic describes the data members that comprise the architecture of a recordset object:

- Field data members
- Parameter data members
- Using m_nFields and m_nParams data members

ⓘ **Note**

This topic applies to objects derived from `CRecordset` in which bulk row fetching has not been implemented. If bulk row fetching is implemented, the architecture is similar. To understand the differences, see **Recordset: Fetching Records in Bulk (ODBC)**.

Sample Class

ⓘ **Note**

The MFC ODBC Consumer wizard is not available in Visual Studio 2019 and later. You can still create a consumer manually.

When you use the MFC ODBC Consumer Wizard from **Add Class** wizard to declare a recordset class derived from `CRecordset`, the resulting class has the general structure shown in the following simple class:

C++

```
class CCourse : public CRecordset
{
public:
   CCourse(CDatabase* pDatabase = NULL);
   ...
   CString m_strCourseID;
   CString m_strCourseTitle;
   CString m_strIDParam;
};
```

At the beginning of the class, the wizard writes a set of field data members. When you create the class, you must specify one or more field data members. If the class is parameterized, as the sample class is (with the data member `m_strIDParam`), you must manually add parameter data members. The wizard does not support adding parameters to a class.

Field Data Members

The most important members of your recordset class are the field data members. For each column you select from the data source, the class contains a data member of the appropriate data type for that column. For example, the sample class shown at the beginning of this topic has two field data members, both of type `CString`, called `m_strCourseID` and `m_strCourseTitle`.

When the recordset selects a set of records, the framework automatically binds the columns of the current record (after the `Open` call, the first record is current) to the field data members of the object. That is, the framework uses the appropriate field data member as a buffer in which to store the contents of a record column.

As the user scrolls to a new record, the framework uses the field data members to represent the current record. The framework refreshes the field data members, replacing the previous record's values. The field data members are also used for updating the current record and for adding new records. As part of the process of updating a record, you specify the update values by assigning values directly to the appropriate field data member or members.

Parameter Data Members

If the class is parameterized, it has one or more parameter data members. A parameterized class lets you base a recordset query on information obtained or calculated at run time.

Typically, the parameter helps narrow the selection, as in the following example. Based on the sample class at the beginning of this topic, the recordset object might execute the following SQL statement:

SQL

```sql
SELECT CourseID, CourseTitle FROM Course WHERE CourseID = ?
```

The "?" is a placeholder for a parameter value that you supply at run time. When you construct the recordset and set its `m_strIDParam` data member to MATH101, the effective SQL statement for the recordset becomes:

SQL

```sql
SELECT CourseID, CourseTitle FROM Course WHERE CourseID = MATH101
```

By defining parameter data members, you tell the framework about parameters in the SQL string. The framework binds the parameter, which lets ODBC know where to get values to substitute for the placeholder. In the example, the resulting recordset contains only the record from the Course table with a CourseID column whose value is MATH101. All specified columns of this record are selected. You can specify as many parameters (and placeholders) as you need.

Note

MFC does nothing itself with the parameters — in particular, it does not perform a text substitution. Instead, MFC tells ODBC where to get the parameter; ODBC retrieves the data and performs the necessary parameterization.

Note

The order of parameters is important. For information about this and more information about parameters, see **Recordset: Parameterizing a Recordset (ODBC)**.

Using m_nFields and m_nParams

When a wizard writes a constructor for your class, it also initializes the m_nFields data member, which specifies the number of field data members in the class. If you add any parameters to your class, you must also add an initialization for the m_nParams data member, which specifies the number of parameter data members. The framework uses these values to work with the data members.

For more information and examples, see Record Field Exchange: Using RFX.

See also

Recordset (ODBC)
Recordset: Declaring a Class for a Table (ODBC)
Record Field Exchange (RFX)

Recordset: Declaring a Class for a Table (ODBC)

Article • 08/02/2021

ⓘ **Note**

The MFC ODBC Consumer wizard is not available in Visual Studio 2019 and later. You can still create a consumer manually.

This topic applies to the MFC ODBC classes.

The most common recordset class opens a single table. To declare a recordset class for a single table, use the MFC ODBC Consumer Wizard from **Add Class** and choose each column you want by naming a corresponding recordset field data member.

Other uses for recordsets include:

- Joining two or more tables.
- Containing the results of a predefined query.

See also

Recordset (ODBC)
Recordset: Creating and Closing Recordsets (ODBC)
Recordset: Declaring a Class for a Predefined Query (ODBC)
Recordset: Performing a Join (ODBC)

Recordset: Creating and Closing Recordsets (ODBC)

Article • 08/02/2021

Note

The MFC ODBC Consumer wizard is not available in Visual Studio 2019 and later. You can still create a consumer manually.

This topic applies to the MFC ODBC classes.

To use a recordset, construct a recordset object and then call its `Open` member function to run the recordset's query and select records. When you finish with the recordset, close and destroy the object.

This topic explains:

- When and how to create a recordset object.
- When and how you can qualify the recordset's behavior by parameterizing, filtering, sorting, or locking it.
- When and how to close a recordset object.

Creating Recordsets at Run Time

Before you can create recordset objects in your program, you typically write application-specific recordset classes. For more information about this preliminary step, see Adding an MFC ODBC Consumer.

Open a dynaset or snapshot object when you need to select records from a data source. The type of object to create depends on what you need to do with the data in your application and on what your ODBC driver supports. For more information, see Dynaset and Snapshot.

To open a recordset

1. Construct an object of your `CRecordset`-derived class.

   You can construct the object on the heap or on the stack frame of a function.

2. Optionally modify the default recordset behavior. For the available options, see Setting Recordset Options.

3. Call the object's Open member function.

In the constructor, pass a pointer to a `CDatabase` object or pass NULL to use a temporary database object that the framework constructs and opens based on the connection string returned by the GetDefaultConnect member function. The `CDatabase` object might already be connected to a data source.

The call to `Open` uses SQL to select records from the data source. The first record selected (if any) is the current record. The values of this record's fields are stored in the recordset object's field data members. If any records were selected, both the `IsBOF` and `IsEOF` member functions return 0.

In your Open call, you can:

- Specify whether the recordset is a dynaset or snapshot. Recordsets open as snapshots by default. Or, you can specify a forward-only recordset, which allows only forward scrolling, one record at a time.

  By default, a recordset uses the default type stored in the `CRecordset` data member `m_nDefaultType`. Wizards write code to initialize `m_nDefaultType` to the recordset type you choose in the wizard. Rather than accepting this default, you can substitute another recordset type.

- Specify a string to replace the default SQL **SELECT** statement that the recordset constructs.

- Specify whether the recordset is read-only or append-only. Recordsets allow full updating by default, but you can limit that to adding new records only or you can disallow all updates.

The following example shows how to open a read-only snapshot object of class `CStudentSet`, an application-specific class:

C++

```
// Construct the snapshot object
CStudentSet rsStudent( NULL );
// Set options if desired, then open the recordset
if(!rsStudent.Open(CRecordset::snapshot, NULL, CRecordset::readOnly))
    return FALSE;
// Use the snapshot to operate on its records...
```

After you call `Open`, use the member functions and data members of the object to work with the records. In some cases, you might want to requery or refresh the recordset to include changes that have occurred on the data source. For more information, see Recordset: Requerying a Recordset (ODBC).

Tip

The connect string you use during development might not be the same connect string that your eventual users need. For ideas about generalizing your application in this regard, see **Data Source: Managing Connections (ODBC)**.

Setting Recordset Options

After you construct your recordset object but before you call `Open` to select records, you might want to set some options to control the recordset's behavior. For all recordsets, you can:

- Specify a filter to constrain record selection.
- Specify a sort order for the records.
- Specify parameters so you can select records using information obtained or calculated at run time.

You can also set the following option if conditions are right:

- If the recordset is updateable and supports locking options, specify the locking method used for updates.

Note

To affect record selection, you must set these options before you call the `Open` member function.

Closing a Recordset

When you finish with your recordset, you must dispose of it and deallocate its memory.

To close a recordset

1. Call its Close member function.
2. Destroy the recordset object.

   If you declared it on the stack frame of a function, the object is destroyed automatically when the object goes out of scope. Otherwise, use the **`delete`** operator.

`Close` frees the recordset's `HSTMT` handle. It does not destroy the C++ object.

See also

Recordset (ODBC)
Recordset: Scrolling (ODBC)
Recordset: Adding, Updating, and Deleting Records (ODBC)

Recordset: Scrolling (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

After you open a recordset, you need to access the records to display values, do calculations, generate reports, and so on. Scrolling lets you move from record to record within your recordset.

This topic explains:

- How to scroll from one record to another in a recordset.
- Under what circumstances scrolling is and is not supported.

Scrolling from One Record to Another

Class `CRecordset` provides the `Move` member functions for scrolling within a recordset. These functions move the current record by rowsets. If you have implemented bulk row fetching, a `Move` operation repositions the recordset by the size of the rowset. If you have not implemented bulk row fetching, a call to a `Move` function repositions the recordset by one record each time. For more information about bulk row fetching, see Recordset: Fetching Records in Bulk (ODBC).

Note

When moving through a recordset, deleted records might not be skipped. For more information, see the **IsDeleted** member function.

In addition to the `Move` functions, `CRecordset` provides member functions for checking whether you have scrolled past the end or ahead of the beginning of your recordset.

To determine whether scrolling is possible in your recordset, call the `CanScroll` member function.

To scroll

1. Forward one record or one rowset: call the MoveNext member function.

2. Backward one record or one rowset: call the MovePrev member function.

3. To the first record in the recordset: call the MoveFirst member function.

4. To the last record in the recordset or to the last rowset: call the MoveLast member function.

5. *N* records relative to the current position: call the Move member function.

To test for the end or the beginning of the recordset

1. Have you scrolled past the last record? Call the IsEOF member function.

2. Have you scrolled ahead of the first record (moving backward)? Call the IsBOF member function.

The following code example uses `IsBOF` and `IsEOF` to detect the limits of a recordset when scrolling in either direction.

```
// Open a recordset; first record is current
CCustSet rsCustSet( NULL );
rsCustSet.Open( );

if( rsCustSet.IsBOF( ) )
    return;
    // The recordset is empty

// Scroll to the end of the recordset, past
// the last record, so no record is current
while ( !rsCustSet.IsEOF( ) )
    rsCustSet.MoveNext( );

// Move to the last record
rsCustSet.MoveLast( );

// Scroll to beginning of the recordset, before
// the first record, so no record is current
while( !rsCustSet.IsBOF( ) )
    rsCustSet.MovePrev( );

// First record is current again
rsCustSet.MoveFirst( );
```

`IsEOF` returns a nonzero value if the recordset is positioned past the last record. `IsBOF` returns a nonzero value if the recordset is positioned ahead of the first record (before all records). In either case, there is no current record to operate on. If you call `MovePrev` when `IsBOF` is already TRUE or call `MoveNext` when `IsEOF` is already TRUE, the framework throws a `CDBException`. You can also use `IsBOF` and `IsEOF` to check for an empty recordset.

For more information about recordset navigation, see Recordset: Bookmarks and Absolute Positions (ODBC).

When Scrolling Is Supported

As originally designed, SQL provided only forward scrolling, but ODBC extends scrolling capabilities. The available level of support for scrolling depends on the ODBC drivers your application works with, your driver's ODBC API conformance level, and whether the ODBC Cursor Library is loaded into memory. For more information, see ODBC and ODBC: The ODBC Cursor Library.

Tip

You can control whether the cursor library is used. See the *bUseCursorLib* and *dwOptions* parameters to **CDatabase::Open**.

Note

Unlike the MFC DAO classes, the MFC ODBC classes do not provide a set of `Find` functions for locating the next (or previous) record that meets specified criteria.

See also

Recordset (ODBC)
CRecordset::CanScroll
CRecordset::CheckRowsetError
Recordset: Filtering Records (ODBC)

Recordset: Bookmarks and Absolute Positions (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

When navigating through a recordset, you often need a way of returning to a particular record. A record's bookmark and absolute position provide two such methods.

This topic explains:

- How to use bookmarks.
- How to set the current record using absolute positions.

Bookmarks in MFC ODBC

A bookmark uniquely identifies a record. When you navigate through a recordset, you cannot always rely on the absolute position of a record because records can be deleted from the recordset. The reliable way to keep track of the position of a record is to use its bookmark. Class `CRecordset` supplies member functions for:

- Getting the bookmark of the current record, so you can save it in a variable (GetBookmark).
- Moving quickly to a given record by specifying its bookmark, which you saved earlier in a variable (SetBookmark).

The following example illustrates how to use these member functions to mark the current record and later return to it:

C++

```cpp
// rs is a CRecordset or
// CRecordset-derived object

CDBVariant varRecordToReturnTo;
rs.GetBookmark( varRecordToReturnTo );

// More code in which you
// move to other records

rs.SetBookmark( varRecordToReturnTo );
```

You do not need to extract the underlying data type from the CDBVariant Class object. Assign the value with `GetBookmark` and return to that bookmark with `SetBookmark`.

ⓘ **Note**

Depending on your ODBC driver and recordset type, bookmarks might not be supported. You can easily determine whether bookmarks are supported by calling **CRecordset::CanBookmark**. Furthermore, if bookmarks are supported, you must explicitly choose to implement them by specifying the `CRecordset::useBookmarks` option in the **CRecordset::Open** member function. You should also check the persistence of bookmarks after certain recordset operations. For example, if you `Requery` a recordset, bookmarks might no longer be valid. Call **CDatabase::GetBookmarkPersistence** to check whether you can safely call `SetBookmark`.

Absolute Positions in MFC ODBC

Besides bookmarks, class `CRecordset` allows you to set the current record by specifying an ordinal position. This is called absolute positioning.

ⓘ **Note**

Absolute positioning is not available on forward-only recordsets. For more information about forward-only recordsets, see **Recordset (ODBC)**.

To move the current record pointer using absolute position, call CRecordset::SetAbsolutePosition. When you pass a value to `SetAbsolutePosition`, the record corresponding to that ordinal position becomes the current record.

ⓘ **Note**

The absolute position of a record is potentially unreliable. If the user deletes records from the recordset, the ordinal position of any subsequent record changes. Bookmarks are the recommended method for moving the current record. For more information, see **Bookmarks in MFC ODBC**.

For more information about recordset navigation, see Recordset: Scrolling (ODBC).

See also

Recordset (ODBC)

Recordset: Filtering Records (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

This topic explains how to filter a recordset so that it selects only a particular subset of the available records. For example, you might want to select only the class sections for a particular course, such as MATH101. A filter is a search condition defined by the contents of a SQL **WHERE** clause. When the framework appends it to the recordset's SQL statement, the **WHERE** clause constrains the selection.

You must establish a recordset object's filter after you construct the object but before you call its `Open` member function (or before you call the `Requery` member function for an existing recordset object whose `Open` member function has been called previously).

To specify a filter for a recordset object

1. Construct a new recordset object (or prepare to call `Requery` for an existing object).

2. Set the value of the object's m_strFilter data member.

   The filter is a null-terminated string that contains the contents of the SQL **WHERE** clause but not the keyword **WHERE**. For example, use:

   ```
   m_pSet->m_strFilter = "CourseID = 'MATH101'";
   ```

   not

   ```
   m_pSet->m_strFilter = "WHERE CourseID = 'MATH101'";
   ```

   Note

   The literal string "MATH101" is shown with single quotation marks above. In the ODBC SQL specification, single quotes are used to denote a character string literal. Check your ODBC driver documentation for the quoting requirements of your DBMS in this situation. This syntax is also discussed further near the end of this topic.

3. Set any other options you need, such as sort order, locking mode, or parameters. Specifying a parameter is especially useful. For information about parameterizing your filter, see Recordset: Parameterizing a Recordset (ODBC).

4. Call `Open` for the new object (or `Requery` for a previously opened object).

Tip

Using parameters in your filter is potentially the most efficient method for retrieving records.

Tip

Recordset filters are useful for **joining** tables and for using **parameters** based on information obtained or calculated at run time.

The recordset selects only those records that meet the search condition you specified. For example, to specify the course filter described above (assuming a variable `strCourseID` currently set, for instance, to "MATH101"), do the following:

```
// Using the recordset pointed to by m_pSet

// Set the filter
m_pSet->m_strFilter = "CourseID = " + strCourseID;

// Run the query with the filter in place
if ( m_pSet->Open( CRecordset::snapshot, NULL, CRecordset::readOnly ) )

// Use the recordset
```

The recordset contains records for all class sections for MATH101.

Notice how the filter string was set in the example above, using a string variable. This is the typical usage. But suppose you wanted to specify the literal value 100 for the course ID. The following code shows how to set the filter string correctly with a literal value:

```
m_strFilter = "StudentID = '100'";   // correct
```

Note the use of single quote characters; if you set the filter string directly, the filter string is **not**:

```
m_strFilter = "StudentID = 100";   // incorrect for some drivers
```

The quoting shown above conforms to the ODBC specification, but some DBMSs might require other quote characters. For more information, see SQL: Customizing Your Recordset's SQL Statement (ODBC).

Note

If you choose to override the recordset's default SQL string by passing your own SQL string to `Open`, you should not set a filter if your custom string has a **WHERE** clause. For more information about overriding the default SQL, see **SQL: Customizing Your Recordset's SQL Statement (ODBC)**.

See also

Recordset (ODBC)
Recordset: Sorting Records (ODBC)
Recordset: How Recordsets Select Records (ODBC)
Recordset: How Recordsets Update Records (ODBC)
Recordset: Locking Records (ODBC)

Recordset: Sorting Records (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

This topic explains how to sort your recordset. You can specify one or more columns on which to base the sort, and you can specify ascending or descending order (**ASC** or **DESC**; **ASC** is the default) for each specified column. For example, if you specify two columns, the records are sorted first on the first column named and then on the second column named. A SQL **ORDER BY** clause defines a sort. When the framework appends the **ORDER BY** clause to the recordset's SQL query, the clause controls the selection's ordering.

You must establish a recordset's sort order after you construct the object but before you call its `Open` member function (or before you call the `Requery` member function for an existing recordset object whose `Open` member function has been called previously).

To specify a sort order for a recordset object

1. Construct a new recordset object (or prepare to call `Requery` for an existing one).

2. Set the value of the object's m_strSort data member.

   The sort is a null-terminated string. It contains the contents of the **ORDER BY** clause but not the keyword **ORDER BY**. For example, use:

   ```
   recordset.m_strSort = "LastName DESC, FirstName DESC";
   ```

   not

   ```
   recordset.m_strSort = "ORDER BY LastName DESC, FirstName DESC";
   ```

3. Set any other options you need, such as a filter, locking mode, or parameters.

4. Call `Open` for the new object (or `Requery` for an existing object).

The selected records are ordered as specified. For example, to sort a set of student records in descending order by last name, then first name, do the following:

```C++
// Construct the recordset
CStudentSet rsStudent( NULL );
// Set the sort
rsStudent.m_strSort = "LastName DESC, FirstName DESC";
// Run the query with the sort in place
rsStudent.Open( );
```

The recordset contains all of the student records, sorted in descending order (Z to A) by last name, then by first name.

> **Note**
>
> If you choose to override the recordset's default SQL string by passing your own SQL string to `Open`, do not set a sort if your custom string has an **ORDER BY** clause.

See also

Recordset (ODBC)
Recordset: Parameterizing a Recordset (ODBC)
Recordset: Filtering Records (ODBC)

Recordset: Parameterizing a Recordset (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

Sometimes you might want to be able to select records at run time, using information you have calculated or obtained from your end user. Recordset parameters let you accomplish that goal.

This topic explains:

- The purpose of a parameterized recordset.
- When and why you might want to parameterize a recordset.
- How to declare parameter data members in your recordset class.
- How to pass parameter information to a recordset object at run time.

Parameterized Recordsets

A parameterized recordset lets you pass parameter information at run time. This has two valuable effects:

- It might result in better execution speed.
- It lets you build a query at run time, based on information not available to you at design time, such as information obtained from your user or calculated at run time.

When you call `Open` to run the query, the recordset uses the parameter information to complete its **SQL SELECT** statement. You can parameterize any recordset.

When to Use Parameters

Typical uses for parameters include:

- Passing run-time arguments to a predefined query.

To pass parameters to a stored procedure, you must specify a complete custom ODBC **CALL** statement — with parameter placeholders — when you call `Open`, overriding the recordset's default SQL statement. For more information, see CRecordset::Open in the *Class Library Reference* and SQL: Customizing Your Recordset's SQL Statement (ODBC) and Recordset: Declaring a Class for a Predefined Query (ODBC).

- Efficiently performing numerous requeries with different parameter information.

  For example, each time your end user looks up information for a particular student in the student registration database, you can specify the student's name or ID as a parameter obtained from the user. Then, when you call your recordset's `Requery` member function, the query selects only that student's record.

  Your recordset's filter string, stored in `m_strFilter`, might look like this:

  ```C++
  "StudentID = ?"
  ```

  Suppose you obtain the student ID in the variable `strInputID`. When you set a parameter to `strInputID` (for example, the student ID 100) the value of the variable is bound to the parameter placeholder represented by the "?" in the filter string.

  Assign the parameter value as follows:

  ```C++
  strInputID = "100";
  ...
  m_strParam = strInputID;
  ```

  You would not want to set up a filter string this way:

  ```C++
  m_strFilter = "StudentID = 100";   // 100 is incorrectly quoted
                                     // for some drivers
  ```

For a discussion of how to use quotes correctly for filter strings, see Recordset: Filtering Records (ODBC).

The parameter value is different each time you requery the recordset for a new student ID.

Tip

Using a parameter is more efficient than simply a filter. For a parameterized recordset, the database must process a SQL **SELECT** statement only once. For a filtered recordset without parameters, the **SELECT** statement must be processed each time you `Requery` with a new filter value.

For more information about filters, see Recordset: Filtering Records (ODBC).

Parameterizing Your Recordset Class

Note

This section applies to objects derived from `CRecordset` in which bulk row fetching has not been implemented. If you are using bulk row fetching, implementing parameters is a similar process. For more information, see **Recordset: Fetching Records in Bulk (ODBC)**.

Before you create your recordset class, determine what parameters you need, what their data types are, and how the recordset uses them.

To parameterize a recordset class

Note

The MFC ODBC Consumer wizard is not available in Visual Studio 2019 and later. You can still create this functionality manually.

1. Run the MFC ODBC Consumer Wizard from **Add Class** to create the class.

2. Specify field data members for the recordset's columns.

3. After the wizard writes the class to a file in your project, go to the .h file and manually add one or more parameter data members to the class declaration. The addition might look something like the following example, part of a snapshot class designed to answer the query "Which students are in the senior class?"

C++

```
class CStudentSet : public CRecordset
{
// Field/Param Data
    CString m_strFirstName;
    CString m_strLastName;
    CString m_strStudentID;
    CString m_strGradYear;

    CString m_strGradYrParam;
};
```

Add your parameter data members after the wizard-generated field data members. The convention is to append the word "Param" to each user-defined parameter name.

4. Modify the DoFieldExchange member function definition in the .cpp file. Add an RFX function call for each parameter data member you added to the class. For information about writing your RFX functions, see Record Field Exchange: How RFX Works. Precede the RFX calls for the parameters with a single call to:

C++

```
pFX->SetFieldType( CFieldExchange::param );
// RFX calls for parameter data members
```

5. In the constructor of your recordset class, increment the count of parameters, `m_nParams`.

For information, see Record Field Exchange: Working with the Wizard Code.

6. When you write the code that creates a recordset object of this class, place a "?" (question mark) symbol in each place in your SQL statement strings where a parameter is to be replaced.

At run time, "?" placeholders are filled, in order, by the parameter values you pass. The first parameter data member set after the SetFieldType call replaces the first "?" in the SQL string, the second parameter data member replaces the second "?", and so on.

Note

Parameter order is important: the order of RFX calls for parameters in your `DoFieldExchange` function must match the order of the parameter placeholders in your SQL string.

Tip

The most likely string to work with is the string you specify (if any) for the class's **m_strFilter** data member, but some ODBC drivers might allow parameters in other SQL clauses.

Passing Parameter Values at Run Time

You must specify parameter values before you call `Open` (for a new recordset object) or `Requery` (for an existing one).

To pass parameter values to a recordset object at run time

1. Construct the recordset object.
2. Prepare a string or strings, such as the `m_strFilter` string, containing the SQL statement, or parts of it. Put "?" placeholders where the parameter information is to go.
3. Assign a run-time parameter value to each parameter data member of the object.
4. Call the `Open` member function (or `Requery`, for an existing recordset).

For example, suppose you want to specify a filter string for your recordset using information obtained at run time. Assume you have constructed a recordset of class `CStudentSet` earlier — called `rsStudents` — and now want to requery it for a particular kind of student information.

C++

```cpp
// Set up a filter string with
// parameter placeholders
rsStudents.m_strFilter = "GradYear <= ?";

// Obtain or calculate parameter values
// to pass--simply assigned here
CString strGradYear = GetCurrentAcademicYear( );

// Assign the values to parameter data members
rsStudents.m_strGradYrParam = strGradYear;

// Run the query
if( !rsStudents.Requery( ) )
    return FALSE;
```

The recordset contains records for those students whose records meet the conditions specified by the filter, which was constructed from run-time parameters. In this case, the recordset contains records for all senior students.

Note

If needed, you can set the value of a parameter data member to Null, using **SetParamNull**. You can likewise check whether a parameter data member is Null, using **IsFieldNull**.

See also

Recordset (ODBC)
Recordset: Adding, Updating, and Deleting Records (ODBC)
Recordset: How Recordsets Select Records (ODBC)

Recordset: Adding, Updating, and Deleting Records (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

> ⓘ **Note**
>
> You can now add records in bulk more efficiently. For more information, see **Recordset: Adding Records in Bulk (ODBC)**.

> ⓘ **Note**
>
> This topic applies to objects derived from `CRecordset` in which bulk row fetching has not been implemented. If you are using bulk row fetching, see **Recordset: Fetching Records in Bulk (ODBC)**.

Updateable snapshots and dynasets allow you to add, edit (update), and delete records. This topic explains:

- How to determine whether your recordset is updatable.
- How to add a new record.
- How to edit an existing record.
- How to delete a record.

For more information about how updates are carried out and how your updates appear to other users, see Recordset: How Recordsets Update Records (ODBC). Normally, when you add, edit, or delete a record, the recordset changes the data source immediately. You can instead batch groups of related updates into transactions. If a transaction is in progress, the update does not become final until you commit the transaction. This allows you to take back or roll back the changes. For information about transactions, see Transaction (ODBC).

The following table summarizes the options available for recordsets with different update characteristics.

Recordset Read/Update Options

Expand table

Type	Read	Edit record	Delete record	Add new (append)
Read-only	Y	N	N	N
Append-only	Y	N	N	Y
Fully updatable	Y	Y	Y	Y

Determining Whether Your Recordset is Updateable

A recordset object is updateable if the data source is updateable and you opened the recordset as updateable. Its updateability also depends on the SQL statement you use, the capabilities of your ODBC driver, and whether the ODBC Cursor Library is in memory. You cannot update a read-only recordset or data source.

To determine whether your recordset is updatable

1. Call the recordset object's CanUpdate member function.

   `CanUpdate` returns a nonzero value if the recordset is updateable.

By default, recordsets are fully updateable (you can perform `AddNew`, `Edit`, and `Delete` operations). But you can also use the appendOnly option to open updateable recordsets. A recordset opened this way allows only the addition of new records with `AddNew`. You cannot edit or delete existing records. You can test whether a recordset is open only for appending by calling the CanAppend member function. `CanAppend` returns a nonzero value if the recordset is either fully updateable or open only for appending.

The following code shows how you might use `CanUpdate` for a recordset object called `rsStudentSet`:

C++
```
if( !rsStudentSet.Open( ) )
    return FALSE;
if( !rsStudentSet.CanUpdate( ) )
{
    AfxMessageBox( "Unable to update the Student recordset." );
    return;
}
```

Caution

When you prepare to update a recordset by calling `Update`, take care that your recordset includes all columns making up the primary key of the table (or all of the columns of any unique index on the table). In some cases, the framework can use only the columns selected in your recordset to identify which record in your table to update. Without all the necessary columns, multiple records might be updated in the table, possibly damaging the referential integrity of the table. In this case, the framework throws exceptions when you call `Update`.

Adding a Record to a Recordset

You can add new records to a recordset if its CanAppend member function returns a nonzero value.

To add a new record to a recordset

1. Make sure the recordset is appendable.

2. Call the recordset object's AddNew member function.

   `AddNew` prepares the recordset to act as an edit buffer. All field data members are set to the special value Null and marked as unchanged so only changed (dirty) values are written to the data source when you call Update.

3. Set the values of the new record's field data members.

   Assign values to the field data members. Those you do not assign are not written to the data source.

4. Call the recordset object's `Update` member function.

   `Update` completes the addition by writing the new record to the data source. For information about happens if you fail to call `Update`, see Recordset: How Recordsets Update Records (ODBC).

For information about how adding records works and about when added records are visible in your recordset, see Recordset: How AddNew, Edit, and Delete Work (ODBC).

The following example shows how to add a new record:

C++

```cpp
if( !rsStudent.Open( ) )
    return FALSE;
if( !rsStudent.CanAppend( ) )
    return FALSE;                      // no field values were set
rsStudent.AddNew( );
rsStudent.m_strName = strName;
rsStudent.m_strCity = strCity;
rsStudent.m_strStreet = strStreet;
if( !rsStudent.Update( ) )
{
    AfxMessageBox( "Record not added; no field values were set." );
    return FALSE;
}
```

Tip

To cancel an `AddNew` or `Edit` call, simply make another call to `AddNew` or `Edit` or call `Move` with the *AFX_MOVE_REFRESH* parameter. Data members are reset to their previous values and you are still in `Edit` or `Add` mode.

Editing a Record in a Recordset

You can edit existing records if your recordset's CanUpdate member function returns a nonzero value.

To edit an existing record in a recordset

1. Make sure the recordset is updateable.
2. Scroll to the record you want to update.
3. Call the recordset object's Edit member function.

   `Edit` prepares the recordset to act as an edit buffer. All field data members are marked so that the recordset can tell later whether they were changed. The new values for changed field data members are written to the data source when you call Update.
4. Set the values of the new record's field data members.

   Assign values to the field data members. Those you do not assign values remain unchanged.
5. Call the recordset object's `Update` member function.

   `Update` completes the edit by writing the changed record to the data source. For information about happens if you fail to call `Update`, see Recordset: How Recordsets Update Records (ODBC).

After you edit a record, the edited record remains the current record.

The following example shows an `Edit` operation. It assumes the user has moved to a record he or she wants to edit.

C++

```cpp
rsStudent.Edit( );
rsStudent.m_strStreet = strNewStreet;
rsStudent.m_strCity = strNewCity;
rsStudent.m_strState = strNewState;
rsStudent.m_strPostalCode = strNewPostalCode;
if( !rsStudent.Update( ) )
{
    AfxMessageBox( "Record not updated; no field values were set." );
    return FALSE;
}
```

Tip

To cancel an `AddNew` or `Edit` call, simply make another call to `AddNew` or `Edit` or call `Move` with the *AFX_MOVE_REFRESH* parameter. Data members are reset to their previous values and you are still in `Edit` or `Add` mode.

Deleting a Record from a Recordset

You can delete records if your recordset's CanUpdate member function returns a nonzero value.

To delete a record

1. Make sure the recordset is updateable.

2. Scroll to the record you want to update.

3. Call the recordset object's Delete member function.

   `Delete` immediately marks the record as deleted, both in the recordset and on the data source.

   Unlike `AddNew` and `Edit`, `Delete` has no corresponding `Update` call.

4. Scroll to another record.

   Note

   When moving through the recordset, deleted records might not be skipped. For more information, see the **IsDeleted** member function.

The following example shows a `Delete` operation. It assumes that the user has moved to a record that the user wants to delete. After `Delete` is called, it is important to move to a new record.

```
rsStudent.Delete( );
rsStudent.MoveNext( );
```

For more information about the effects of the `AddNew`, `Edit`, and `Delete` member functions, see Recordset: How Recordsets Update Records (ODBC).

See also

Recordset (ODBC)
Recordset: Locking Records (ODBC)

Recordset: Locking Records (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

This topic explains:

- The kinds of record locking available.
- How to lock records in your recordset during updates.

When you use a recordset to update a record on the data source, your application can lock the record so no other user can update the record at the same time. The state of a record updated by two users at the same time is undefined unless the system can guarantee that two users cannot update a record simultaneously.

Note

This topic applies to objects derived from `CRecordset` in which bulk row fetching has not been implemented. If you have implemented bulk row fetching, some of the information does not apply. For example, you cannot call the `Edit` and `Update` member functions. For more information about bulk row fetching, see **Recordset: Fetching Records in Bulk (ODBC)**.

Record-Locking Modes

The database classes provide two record-locking modes:

- Optimistic locking (the default)
- Pessimistic locking

Updating a record occurs in three steps:

1. You begin the operation by calling the Edit member function.
2. You change the appropriate fields of the current record.
3. You end the operation — and normally commit the update — by calling the

Update member function.

Optimistic locking locks the record on the data source only during the `Update` call. If you use optimistic locking in a multiuser environment, the application should handle an `Update` failure condition. Pessimistic locking locks the record as soon as you call `Edit` and does not release it until you call `Update` (failures are indicated through the `CDBException` mechanism, not by a value of FALSE returned by `Update`). Pessimistic locking has a potential performance penalty for other users, because concurrent access to the same record might have to wait until completion of your application's `Update` process.

Locking Records in Your Recordset

If you want to change a recordset object's locking mode from the default, you must change the mode before you call `Edit`.

To change the current locking mode for your recordset

1. Call the SetLockingMode member function, specifying either `CRecordset::pessimistic` or `CRecordset::optimistic`.

The new locking mode remains in effect until you change it again or the recordset is closed.

Note

Relatively few ODBC drivers currently support pessimistic locking.

See also

Recordset (ODBC)
Recordset: Performing a Join (ODBC)
Recordset: Adding, Updating, and Deleting Records (ODBC)

Recordset: Requerying a Recordset (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

This topic explains how you can use a recordset object to requery (that is, refresh) itself from the database and when you might want to do that with the Requery member function.

The principal reasons for requerying a recordset are to:

- Bring the recordset up to date with respect to records added by you or by other users and records deleted by other users (those you delete are already reflected in the recordset).
- Refresh the recordset based on changing parameter values.

Bringing the Recordset Up to Date

Frequently, you will want to requery your recordset object to bring it up to date. In a multiuser database environment, other users can make changes to the data during the life of your recordset. For more information about when your recordset reflects changes made by other users and when other users' recordsets reflect your changes, see Recordset: How Recordsets Update Records (ODBC) and Dynaset.

Requerying Based on New Parameters

Another frequent — and equally important — use of Requery is to select a new set of records based on changing parameter values.

Tip

Query speed is probably significantly faster if you call `Requery` with changing parameter values than if you call `Open` again.

Requerying Dynasets vs. Snapshots

Because dynasets are meant to present a set of records with dynamic up-to-date data, you want to requery dynasets often if you want to reflect other users' additions. Snapshots, on the other hand, are useful because you can safely rely on their static contents while you prepare reports, calculate totals, and so on. Still, you might sometimes want to requery a snapshot as well. In a multiuser environment, snapshot data might lose synchronization with the data source as other users change the database.

To requery a recordset object

1. Call the Requery member function of the object.

Alternatively, you can close and reopen the original recordset. In either case, the new recordset represents the current state of the data source.

For an example, see Record Views: Filling a List Box from a Second Recordset.

Tip

To optimize `Requery` performance, avoid changing the recordset's **filter** or **sort**. Change only the parameter value before calling `Requery`.

If the `Requery` call fails, you can retry the call; otherwise, your application should terminate gracefully. A call to `Requery` or `Open` might fail for any of a number of reasons. Perhaps a network error occurs; or, during the call, after the existing data is released but before the new data is obtained, another user might get exclusive access; or the table on which your recordset depends could be deleted.

See also

Recordset (ODBC)
Recordset: Dynamically Binding Data Columns (ODBC)
Recordset: Creating and Closing Recordsets (ODBC)

Recordset: Performing a Join (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

What a Join Is

The join operation, a common data-access task, lets you work with data from more than one table using a single recordset object. Joining two or more tables yields a recordset that can contain columns from each table, but appears as a single table to your application. Sometimes the join uses all columns from all tables, but sometimes the SQL **SELECT** clause in a join uses only some of the columns from each table. The database classes support read-only joins but not updateable joins.

To select records containing columns from joined tables, you need the following items:

- A table list containing the names of all tables being joined.
- A column list containing the names of all participating columns. Columns with the same name but from different tables are qualified by the table name.
- A filter (SQL **WHERE** clause) that specifies the columns on which the tables are joined. This filter takes the form "Table1.KeyCol = Table2.KeyCol" and actually accomplishes the join.

You can join more than two tables in the same way by equating multiple pairs of columns, each pair joined by the SQL keyword **AND**.

See also

Recordset (ODBC)
Recordset: Declaring a Class for a Predefined Query (ODBC)
Recordset: Declaring a Class for a Table (ODBC)
Recordset: Requerying a Recordset (ODBC)

Recordset: Declaring a Class for a Predefined Query (ODBC)

Article • 08/02/2021

Note

The MFC ODBC Consumer wizard is not available in Visual Studio 2019 and later. You can still create a consumer manually.

This topic applies to the MFC ODBC classes.

This topic explains how to create a recordset class for a predefined query (sometimes called a stored procedure, as in Microsoft SQL Server).

Note

This topic applies to objects derived from `CRecordset` in which bulk row fetching has not been implemented. If bulk row fetching is implemented, the process is very similar. To understand the differences between recordsets that implement bulk row fetching and those that do not, see **Recordset: Fetching Records in Bulk (ODBC)**.

Some database management systems (DBMSs) allow you to create a predefined query and call it from your programs like a function. The query has a name, might take parameters, and might return records. The procedure in this topic describes how to call a predefined query that returns records (and perhaps takes parameters).

The database classes do not support updating predefined queries. The difference between a snapshot predefined query and a dynaset predefined query is not updateability but whether changes made by other users (or other recordsets in your program) are visible in your recordset.

Tip

You do not need a recordset to call a predefined query that does not return records. Prepare the SQL statement as described below, but execute it by calling the `CDatabase` member function **ExecuteSQL**.

You can create a single recordset class to manage calling a predefined query, but you must do some of the work yourself. The wizards do not support creating a class specifically for this purpose.

To create a class for calling a predefined query (stored procedure)

1. Use the MFC ODBC Consumer Wizard from **Add Class** to create a recordset class for the table that contributes the most columns returned by the query. This gives you a head start.

2. Manually add field data members for any columns of any tables that the query returns but that the wizard did not create for you.

   For example, if the query returns three columns each from two additional tables, add six field data members (of the appropriate data types) to the class.

3. Manually add RFX function calls in the DoFieldExchange member function of the class, one corresponding to the data type of each added field data member.

   C++

   ```
   Immediately before these RFX calls, call <MSHelp:link key-
   words="_mfc_CFieldExchange.3a3a.SetFieldType"
   TABINDEX="0">SetFieldType</MSHelp:link>, as shown here:
   pFX->SetFieldType( CFieldExchange::outputColumn );
   ```

   ⓘ **Note**

   You must know the data types and the order of columns returned in the result set. The order of RFX function calls in `DoFieldExchange` must match the order of result set columns.

4. Manually add initializations for the new field data members in the recordset class constructor.

   You must also increment the initialization value for the m_nFields data member. The wizard writes the initialization, but it only covers the field data members it adds for you. For example:

```C++
m_nFields += 6;
```

Some data types should not be initialized here, for example, `CLongBinary` or byte arrays.

5. If the query takes parameters, add a parameter data member for each parameter, an RFX function call for each, and an initialization for each.

6. You must increment `m_nParams` for each added parameter, as you did `m_nFields` for added fields in step 4 of this procedure. For more information, see Recordset: Parameterizing a Recordset (ODBC).

7. Manually write a SQL statement string with the following form:

```
{CALL proc-name [(? [, ?]...)]}
```

where **CALL** is an ODBC keyword, **proc-name** is the name of the query as it is known on the data source, and the "?" items are placeholders for the parameter values you supply to the recordset at run time (if any). The following example prepares a placeholder for one parameter:

```
CString mySQL = "{CALL Delinquent_Accts (?)}";
```

8. In the code that opens the recordset, set the values of the recordset's parameter data members and then call the `Open` member function, passing your SQL string for the *lpszSQL* parameter. Or instead, replace the string returned by the `GetDefaultSQL` member function in your class.

The following examples show the procedure for calling a predefined query, named `Delinquent_Accts`, which takes one parameter for a sales district number. This query returns three columns: `Acct_No`, `L_Name`, `Phone`. All columns are from the Customers table.

The following recordset specifies field data members for the columns the query returns and a parameter for the sales district number requested at run time.

C++

```
class CDelinquents : public CRecordset
{
// Field/Param Data
    LONG m_lAcct_No;
    CString m_strL_Name;
    CString m_strPhone;
    LONG m_lDistParam;
    // ...
};
```

This class declaration is as the wizard writes it, except for the `m_lDistParam` member added manually. Other members are not shown here.

The next example shows the initializations for the data members in the `CDelinquents` constructor.

C++

```
CDelinquents::CDelinquents(CDatabase* pdb)
   : CRecordset(pdb)
{
    // Wizard-generated params:
    m_lAcct_No = 0;
    m_strL_Name = "";
    m_strPhone = "";
    m_nFields = 3;
    // User-defined params:
    m_nParams = 1;
    m_lDistParam = 0;
}
```

Note the initializations for m_nFields and m_nParams. The wizard initializes `m_nFields`; you initialize `m_nParams`.

The next example shows the RFX functions in `CDelinquents::DoFieldExchange`:

C++

```
void CDelinquents::DoFieldExchange(CFieldExchange* pFX)
{
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, "Acct_No", m_lAcct_No);
    RFX_Text(pFX, "L_Name", m_strL_Name);
    RFX_Text(pFX, "Phone", m_strPhone);
    pFX->SetFieldType(CFieldExchange::param);
    RFX_Long(pFX, "Dist_No", m_lDistParam);
}
```

Besides making the RFX calls for the three returned columns, this code manages binding the parameter you pass at run time. The parameter is keyed to the `Dist_No` (district number) column.

The next example shows how to set up the SQL string and how to use it to open the recordset.

C++

```
// Construct a CDelinquents recordset object
CDelinquents rsDel( NULL );
CString strSQL = "{CALL Delinquent_Accts (?)}"
// Specify a parameter value (obtained earlier from the user)
rsDel.m_lDistParam = lDistrict;
// Open the recordset and run the query
if( rsDel.Open( CRecordset::snapshot, strSQL ) )
    // Use the recordset ...
```

This code constructs a snapshot, passes it a parameter obtained earlier from the user, and calls the predefined query. When the query runs, it returns records for the specified sales district. Each record contains columns for the account number, customer's last name, and customer's phone number.

Tip

You might want to handle a return value (output parameter) from a stored procedure. For more information and an example, see CFieldExchange::SetFieldType.

See also

Recordset (ODBC)
Recordset: Requerying a Recordset (ODBC)
Recordset: Declaring a Class for a Table (ODBC)
Recordset: Performing a Join (ODBC)

Recordset: Dynamically Binding Data Columns (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

Recordsets manage binding table columns that you specify at design time, but there are cases when you might want to bind columns that were unknown to you at design time. This topic explains:

- When you might want to bind columns dynamically to a recordset.
- How to bind columns dynamically at run time.

ⓘ **Note**

This topic applies to objects derived from `CRecordset` in which bulk row fetching has not been implemented. The techniques described generally are not recommended if you are using bulk row fetching. For more information about bulk row fetching, see **Recordset: Fetching Records in Bulk (ODBC)**.

When You Might Bind Columns Dynamically

ⓘ **Note**

The MFC ODBC Consumer wizard is not available in Visual Studio 2019 and later. You can still create a consumer manually.

At design time, the MFC Application Wizard or MFC ODBC Consumer Wizard (from **Add Class**) creates recordset classes based on the known tables and columns on your data source. Databases can change between when you design them and later when your application uses those tables and columns at run time. You or another user might add or drop a table or add or drop columns from a table that your application's recordset relies on. This probably is not a concern for all data-access applications, but if it is for yours, how can you cope with changes in the database schema, other than by redesigning and recompiling? The purpose of this topic is to answer that question.

This topic describes the most common case in which you might bind columns dynamically — having begun with a recordset based on a known database schema, you want to handle additional columns at run time. The topic further assumes that the additional columns map to `CString` field data members, the most common case, although suggestions are supplied to help you manage other data types.

With a small amount of extra code, you can:

- Determine what columns are available at run time.
- Bind additional columns to your recordset dynamically, at run time.

Your recordset still contains data members for the columns you knew about at design time. It also contains a small amount of extra code that dynamically determines whether any new columns have been added to your target table and, if so, binds these new columns to dynamically allocated storage (rather than to recordset data members).

This topic does not cover other dynamic binding cases, such as dropped tables or columns. For those cases, you need to use ODBC API calls more directly. For information, see the ODBC Programmer's Reference.

How to Bind Columns Dynamically

To bind columns dynamically, you must know (or be able to determine) the names of the additional columns. You must also allocate storage for the additional field data members, specify their names and their types, and specify the number of columns you are adding.

The following discussion mentions two different recordsets. The first is the main recordset that selects records from the target table. The second is a special column recordset used to get information about the columns in your target table.

General Process

At the most general level, you follow these steps:

1. Construct your main recordset object.

Optionally, pass a pointer to an open `CDatabase` object or be able to supply connection information to the column recordset in some other way.

2. Take steps to add columns dynamically.

   See the process described in Adding the Columns below.

3. Open your main recordset.

   The recordset selects records and uses record field exchange (RFX) to bind both the static columns (those mapped to recordset field data members) and the dynamic columns (mapped to extra storage that you allocate).

Adding the Columns

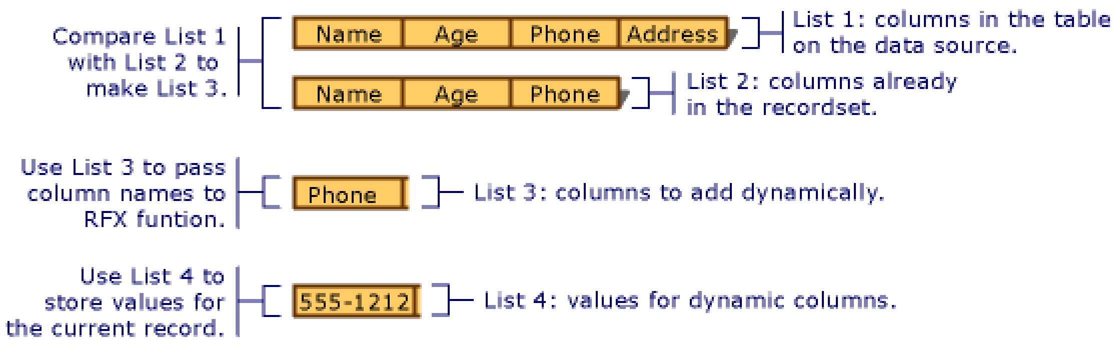
Dynamically binding added columns at run time requires the following steps:

1. Determine at run time what columns are in the target table. Extract from that information a list of the columns that have been added to the table since your recordset class was designed.

   A good approach is to use a column recordset class designed to query the data source for column information for the target table (such as column name and data type).

2. Provide storage for the new field data members. Because your main recordset class does not have field data members for unknown columns, you must provide a place to store the names, result values, and possibly data type information (if the columns are different data types).

   One approach is to build one or more dynamic lists, one for the new columns' names, another for their result values, and a third for their data types (if necessary). These lists, particularly the value list, provide the information and the necessary storage for binding. The following figure illustrates building the lists.


Compare List 1 with List 2 to make List 3.
Name
Age
Phone
Address
List 1: columns in the table on the data source.
Name
Age
Phone
List 2: columns already in the recordset.
Use List 3 to pass column names to RFX funtion.
Phone
List 3: columns to add dynamically.
Use List 4 to store values for the current record.
555-1212
List 4: values for dynamic columns.

Building Lists of Columns to Bind Dynamically

3. Add an RFX function call in your main recordset's `DoFieldExchange` function for each added column. These RFX calls do the work of fetching a record, including the additional columns, and binding the columns to recordset data members or to your dynamically supplied storage for them.

   One approach is to add a loop to your main recordset's `DoFieldExchange` function that loops through your list of new columns, calling the appropriate RFX function for each column in the list. On each RFX call, pass a column name from the column name list and a storage location in the corresponding member of the result value list.

Lists of Columns

The four lists you need to work with are shown in the following table.

Expand table

List	Description
Current-Table-Columns	(List 1 in the illustration) A list of the columns currently in the table on the data source. This list might match the list of columns currently bound in your recordset.
Bound-Recordset-Columns	(List 2 in the illustration) A list of the columns bound in your recordset. These columns already have RFX statements in your `DoFieldExchange` function.
Columns-To-Bind-Dynamically	(List 3 in the illustration) A list of columns in the table but not in your recordset. These are the columns you want to bind dynamically.
Dynamic-Column-Values	(List 4 in the illustration) A list containing storage for the values retrieved from the columns you bind dynamically. Elements of this list correspond to those in Columns-to-Bind-Dynamically, one to one.

Building Your Lists

With a general strategy in mind, you can turn to the details. The procedures in the rest of this topic show you how to build the lists shown in Lists of Columns. The procedures guide you through:

- Determining the names of columns not in your recordset.
- Providing dynamic storage for columns newly added to the table.
- Dynamically adding RFX calls for new columns.

Determining Which Table Columns Are Not in Your Recordset

Build a list (Bound-Recordset-Columns, as in List 2 in the illustration) that contains a list of the columns already bound in your main recordset. Then build a list (Columns-to-Bind-Dynamically, derived from Current-Table-Columns and Bound-Recordset-Columns) that contains column names that are in the table on the data source but not in your main recordset.

To determine the names of columns not in the recordset (Columns-to-Bind-Dynamically)

1. Build a list (Bound-Recordset-Columns) of the columns already bound in your main recordset.

   One approach is to create Bound-Recordset-Columns at design time. You can visually examine the RFX function calls in the recordset's `DoFieldExchange` function to get these names. Then, set up your list as an array initialized with the names.

   For example, the illustration shows Bound-Recordset-Columns (List 2) with three elements. Bound-Recordset-Columns is missing the Phone column shown in Current-Table-Columns (List 1).

2. Compare Current-Table-Columns and Bound-Recordset-Columns to build a list (Columns-to-Bind-Dynamically) of the columns not already bound in your main recordset.

One approach is to loop through your list of columns in the table at run time (Current-Table-Columns) and your list of columns already bound in your recordset (Bound-Recordset-Columns) in parallel. Into Columns-to-Bind-Dynamically put any names in Current-Table-Columns that do not appear in Bound-Recordset-Columns.

For example, the illustration shows Columns-to-Bind-Dynamically (List 3) with one element: the Phone column found in Current-Table-Columns (List 1) but not in Bound-Recordset-Columns (List 2).

3. Build a list of Dynamic-Column-Values (as in List 4 in the illustration) in which to store the data values corresponding to each column name stored in your list of columns to bind dynamically (Columns-to-Bind-Dynamically).

   The elements of this list play the role of new recordset field data members. They are the storage locations to which the dynamic columns are bound. For descriptions of the lists, see Lists of Columns.

Providing Storage for the New Columns

Next, set up storage locations for the columns to be bound dynamically. The idea is to provide a list element in which to store each column's value. These storage locations parallel the recordset member variables, which store the normally bound columns.

To provide dynamic storage for new columns (Dynamic-Column-Values)

1. Build Dynamic-Column-Values, parallel to Columns-to-Bind-Dynamically, to contain the value of the data in each column.

   For example, the illustration shows Dynamic-Column-Values (List 4) with one element: a `CString` object containing the actual phone number for the current record: "555-1212".

   In the most common case, Dynamic-Column-Values has elements of type `CString`. If you are dealing with columns of varying data types, you need a list that can contain elements of a variety of types.

The result of the preceding procedures is two main lists: Columns-to-Bind-Dynamically containing the names of columns and Dynamic-Column-Values containing the values in the columns for the current record.

Tip

If the new columns are not all of the same data type, you might want an extra parallel list containing items that somehow define the type of each corresponding element in the column list. (You can use the values AFX_RFX_BOOL, AFX_RFX_BYTE, and so on, for this if you want. These constants are defined in AFXDB.H.) Choose a list type based on how you represent the column data types.

Adding RFX Calls to Bind the Columns

Finally, arrange for the dynamic binding to occur by placing RFX calls for the new columns in your `DoFieldExchange` function.

To dynamically add RFX calls for new columns

1. In your main recordset's `DoFieldExchange` member function, add code that loops through your list of new columns (Columns-to-Bind-Dynamically). In each loop, extract a column name from Columns-to-Bind-Dynamically and a result value for the column from Dynamic-Column-Values. Pass these items to an RFX function call appropriate to the data type of the column. For descriptions of the lists, see Lists of Columns.

In the common case, in your `RFX_Text` function calls you extract `CString` objects from the lists, as in the following lines of code, where Columns-to-Bind-Dynamically is a `CStringList` called `m_listName` and Dynamic-Column-Values is a `CStringList` called `m_listValue`:

C++

```cpp
RFX_Text( pFX,
            m_listName.GetNext( posName ),
            m_listValue.GetNext( posValue ));
```

For more information about RFX functions, see Macros and Globals in the *Class Library Reference*.

Tip

If the new columns are different data types, use a switch statement in your loop to call the appropriate RFX function for each type.

When the framework calls `DoFieldExchange` during the `Open` process to bind columns to the recordset, the RFX calls for the static columns bind those columns. Then your loop repeatedly calls RFX functions for the dynamic columns.

See also

Recordset (ODBC)
Recordset: Working with Large Data Items (ODBC)

Recordset: Fetching Records in Bulk (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

Class `CRecordset` provides support for bulk row fetching, which means that multiple records can be retrieved at once during a single fetch, rather than retrieving one record at a time from the data source. You can implement bulk row fetching only in a derived `CRecordset` class. The process of transferring data from the data source to the recordset object is called bulk record field exchange (Bulk RFX). Note that if you are not using bulk row fetching in a `CRecordset`-derived class, data is transferred via record field exchange (RFX). For more information, see Record Field Exchange (RFX).

This topic explains:

- How CRecordset supports bulk row fetching.
- Some special considerations when using bulk row fetching.
- How to implement bulk record field exchange.

How CRecordset Supports Bulk Row Fetching

Before opening your recordset object, you can define a rowset size with the `SetRowsetSize` member function. The rowset size specifies how many records should be retrieved during a single fetch. When bulk row fetching is implemented, the default rowset size is 25. If bulk row fetching is not implemented, the rowset size remains fixed at 1.

After you have initialized the rowset size, call the Open member function. Here you must specify the `CRecordset::useMultiRowFetch` option of the *dwOptions* parameter to implement bulk row fetching. You can additionally set the `CRecordset::userAllocMultiRowBuffers` option. The bulk record field exchange mechanism uses arrays to store the multiple rows of data retrieved during a fetch. These storage buffers can be allocated automatically by the framework or you can allocate them manually. Specifying the `CRecordset::userAllocMultiRowBuffers` option means that you will do the allocation.

The following table lists the member functions provided by `CRecordset` to support bulk row fetching.

Expand table

Member function	Description
CheckRowsetError	Virtual function that handles any errors that occur during fetching.
DoBulkFieldExchange	Implements bulk record field exchange. Called automatically to transfers multiple rows of data from the data source to the recordset object.
GetRowsetSize	Retrieves the current setting for the rowset size.
GetRowsFetched	Tells how many rows were actually retrieved after a given fetch. In most cases, this is the rowset size, unless an incomplete rowset was fetched.
GetRowStatus	Returns a fetch status for a particular row within a rowset.
RefreshRowset	Refreshes the data and status of a particular row within a rowset.
SetRowsetCursorPosition	Moves the cursor to a particular row within a rowset.
SetRowsetSize	Virtual function that changes the setting for the rowset size to the specified value.

Special Considerations

Although bulk row fetching is a performance gain, certain features operate differently. Before you decide to implement bulk row fetching, consider the following:

- The framework automatically calls the `DoBulkFieldExchange` member function to transfer data from the data source to the recordset object. However, data is not transferred from the recordset back to the data source. Calling the `AddNew`, `Edit`, `Delete`, or `Update` member functions results in a failed assertion. Although `CRecordset` currently does not provide a mechanism for updating bulk rows of data, you can write your own functions by using the ODBC API function `SQLSetPos`. For more information about `SQLSetPos`, see the ODBC Programmer's Reference.

- The member functions `IsDeleted`, `IsFieldDirty`, `IsFieldNull`, `IsFieldNullable`, `SetFieldDirty`, and `SetFieldNull` cannot be used on recordsets that implement bulk row fetching. However, you can call `GetRowStatus` in place of `IsDeleted`, and `GetODBCFieldInfo` in place of `IsFieldNullable`.

- The `Move` operations repositions your recordset by rowset. For example, suppose you open a recordset that has 100 records with an initial rowset size of 10. `Open` fetches rows 1 through 10, with the current record positioned on row 1. A call to `MoveNext` fetches the next rowset, not the next row. This rowset consists of rows 11 through 20, with the current record positioned on row 11. Note that `MoveNext` and `Move( 1 )` are not equivalent when bulk row fetching is implemented. `Move( 1 )` fetches the rowset that begins 1 row from the current record. In this example, calling `Move( 1 )` after calling `Open` fetches the rowset consisting of rows 2 through 11, with the current record positioned on row 2. For more information, see the Move member function.

- Unlike record field exchange, the wizards do not support bulk record field exchange. This means that you must manually declare your field data members and manually override `DoBulkFieldExchange` by writing calls to the Bulk RFX functions. For more information, see Record Field Exchange Functions in the *Class Library Reference*.

How to Implement Bulk Record Field Exchange

Bulk record field exchange transfers a rowset of data from the data source to the recordset object. The Bulk RFX functions use arrays to store this data, as well as arrays to store the length of each data item in the rowset. In your class definition, you must define your field data members as pointers to access the arrays of data. In addition, you must define a set of pointers to access the arrays of lengths. Any parameter data members should not be declared as pointers; declaring parameter data members when using bulk record field exchange is the same as declaring them when using record field exchange. The following code shows a simple example:

C++

```cpp
class MultiRowSet : public CRecordset
{
public:
   // Field/Param Data
      // field data members
      long* m_rgID;
      LPSTR m_rgName;

      // pointers for the lengths
      // of the field data
      long* m_rgIDLengths;
      long* m_rgNameLengths;

      // input parameter data member
      CString m_strNameParam;

   .
   .
   .
}
```

You can either allocate these storage buffers manually or have the framework do the allocation. To allocate the buffers yourself, you must specify the `CRecordset::userAllocMultiRowBuffers` option of the *dwOptions* parameter in the `Open` member function. Be sure to set the sizes of the arrays at least equal to the rowset size. If you want to have the framework do the allocation, you should initialize your pointers to NULL. This is typically done in the recordset object's constructor:

C++

```
MultiRowSet::MultiRowSet( CDatabase* pDB )
   : CRecordset( pDB )
{
   m_rgID = NULL;
   m_rgName = NULL;
   m_rgIDLengths = NULL;
   m_rgNameLengths = NULL;
   m_strNameParam = "";

   m_nFields = 2;
   m_nParams = 1;

   .
   .
   .
}
```

Finally, you must override the `DoBulkFieldExchange` member function. For the field data members, call the Bulk RFX functions; for any parameter data members, call the RFX functions. If you opened the recordset by passing a SQL statement or stored procedure to `Open`, the order in which you make the Bulk RFX calls must correspond to the order of the columns in the recordset; similarly, the order of the RFX calls for parameters must correspond to the order of parameters in the SQL statement or stored procedure.

C++

```cpp
void MultiRowSet::DoBulkFieldExchange( CFieldExchange* pFX )
{
   // call the Bulk RFX functions
   // for field data members
   pFX->SetFieldType( CFieldExchange::outputColumn );
   RFX_Long_Bulk( pFX, _T( "[colRecID]" ),
                  &m_rgID, &m_rgIDLengths );
   RFX_Text_Bulk( pFX, _T( "[colName]" ),
                  &m_rgName, &m_rgNameLengths, 30 );

   // call the RFX functions for
   // for parameter data members
   pFX->SetFieldType( CFieldExchange::inputParam );
   RFX_Text( pFX, "NameParam", m_strNameParam );
}
```

Note

You must call the `Close` member function before your derived `CRecordset` class goes out of scope. This ensures that any memory allocated by the framework are freed. It is good programming practice to always explicitly call `Close`, regardless of whether you have implemented bulk row fetching.

For more information about record field exchange (RFX), see Record Field Exchange: How RFX Works. For more information about using parameters, see CFieldExchange::SetFieldType and Recordset: Parameterizing a Recordset (ODBC).

See also

Recordset (ODBC)
CRecordset::m_nFields
CRecordset::m_nParams

Recordset: Working with Large Data Items (ODBC)

Article • 08/02/2021

This topic applies to both the MFC ODBC classes and the MFC DAO classes.

Note

If you are using the MFC DAO classes, manage your large data items with class **CByteArray** rather than class **CLongBinary**. If you are using the MFC ODBC classes with bulk row fetching, use `CLongBinary` rather than `CByteArray`. For more information about bulk row fetching, see **Recordset: Fetching Records in Bulk (ODBC)**.

Suppose your database can store large pieces of data, such as bitmaps (employee photographs, maps, pictures of products, OLE objects, and so on). This kind of data is often referred to as a Binary Large Object (or BLOB) because:

- Each field value is large.
- Unlike numbers and other simple data types, it has no predictable size.
- The data is formless from the perspective of your program.

This topic explains what support the database classes provide for working with such objects.

Managing Large Objects

Recordsets have two ways to solve the special difficulty of managing binary large objects. You can use class CByteArray or you can use class CLongBinary. In general, `CByteArray` is the preferred way to manage large binary data.

`CByteArray` requires more overhead than `CLongBinary` but is more capable, as described in The CByteArray Class. `CLongBinary` is described briefly in The CLongBinary Class.

For detailed information about using `CByteArray` to work with large data items, see

Technical Note 45.

CByteArray Class

`CByteArray` is one of the MFC collection classes. A `CByteArray` object stores a dynamic array of bytes — the array can grow if needed. The class provides fast access by index, as with built-in C++ arrays. `CByteArray` objects can be serialized and dumped for diagnostic purposes. The class supplies member functions for getting and setting specified bytes, inserting and appending bytes, and removing one byte or all bytes. These facilities make parsing the binary data easier. For example, if the binary object is an OLE object, you might have to work through some header bytes to reach the actual object.

Using CByteArray in Recordsets

By giving a field data member of your recordset the type `CByteArray`, you provide a fixed base from which RFX can manage the transfer of such an object between your recordset and the data source and through which you can manipulate the data inside the object. RFX needs a specific site for retrieved data, and you need a way to access the underlying data.

For detailed information about using `CByteArray` to work with large data items, see Technical Note 45.

CLongBinary Class

A CLongBinary object is a simple shell around an `HGLOBAL` handle to a block of storage allocated on the heap. When it binds a table column containing a binary large object, RFX allocates the `HGLOBAL` handle when it needs to transfer the data to the recordset and stores the handle in the `CLongBinary` field of the recordset.

In turn, you use the `HGLOBAL` handle, `m_hData`, to work with the data itself, operating on it as you would on any handle data. This is where CByteArray adds capabilities.

ⓧ **Caution**

CLongBinary objects cannot be used as parameters in function calls. In addition, their implementation, which calls `::SQLGetData`, necessarily slows scrolling performance for a scrollable snapshot. This might also be true when you use an `::SQLGetData` call yourself to retrieve dynamic schema columns.

See also

Recordset (ODBC)
Recordset: Obtaining SUMs and Other Aggregate Results (ODBC)
Record Field Exchange (RFX)

Recordset: Obtaining SUMs and Other Aggregate Results (ODBC)

Article • 08/02/2021

Note

The MFC ODBC Consumer wizard is not available in Visual Studio 2019 and later. You can still create a consumer manually.

This topic applies to the MFC ODBC classes.

This topic explains how to obtain aggregate results using the following SQL keywords:

- **SUM** Calculates the total of the values in a column with a numeric data type.
- **MIN** Extracts the smallest value in a column with a numeric data type.
- **MAX** Extracts the largest value in a column with a numeric data type.
- **AVG** Calculates an average value of all the values in a column with a numeric data type.
- **COUNT** Counts the number of records in a column of any data type.

You use these SQL functions to obtain statistical information about the records in a data source rather than to extract records from the data source. The recordset that is created usually consists of a single record (if all columns are aggregates) that contains a value. (There might be more than one record if you used a **GROUP BY** clause.) This value is the result of the calculation or extraction performed by the SQL function.

Tip

To add a SQL **GROUP BY** clause (and possibly a **HAVING** clause) to your SQL statement, append it to the end of `m_strFilter`. For example:

```
m_strFilter = "sales > 10 GROUP BY SALESPERSON_ID";
```

You can limit the number of records you use to obtain aggregate results by filtering and sorting the columns.

Caution

Some aggregation operators return a different data type from the columns over which they are aggregating.

- **SUM** and **AVG** might return the next larger data type (for example, calling with `int` returns **LONG** or `double`).
- **COUNT** usually returns **LONG** regardless of target column type.
- **MAX** and **MIN** return the same data type as the columns they calculate.

  For example, the **Add Class** wizard creates `long m_lSales` to accommodate a Sales column, but you need to replace this with a `double m_dblSumSales` data member to accommodate the aggregate result. See the following example.

To obtain an aggregate result for a recordset

1. Create a recordset as described in Adding an MFC ODBC Consumer containing the columns from which you want to obtain aggregate results.
2. Modify the DoFieldExchange function for the recordset. Replace the string representing the column name (the second argument of the RFX function calls) with a string representing the aggregation function on the column. For example, replace:

   ```
   RFX_Long(pFX, "Sales", m_lSales);
   ```

   with:

```
RFX_Double(pFX, "Sum(Sales)", m_dblSumSales)
```

3. Open the recordset. The result of the aggregation operation is left in `m_dblSumSales`.

> **Note**
>
> The wizard actually assigns data member names without Hungarian prefixes. For example, the wizard would produce `m_Sales` for a Sales column, rather than the `m_lSales` name used earlier for illustration.

If you are using a CRecordView class to view the data, you have to change the DDX function call to display the new data member value; in this case, changing it from:

```
DDX_FieldText(pDX, IDC_SUMSALES, m_pSet->m_lSales, m_pSet);
```

To:

```
DDX_FieldText(pDX, IDC_SUMSALES, m_pSet->m_dblSumSales, m_pSet);
```

See also

Recordset (ODBC)
Recordset: How Recordsets Select Records (ODBC)

Recordset: How Recordsets Select Records (ODBC)

Article • 08/02/2021

ⓘ **Note**

The MFC ODBC Consumer wizard is not available in Visual Studio 2019 and later. You can still create a consumer manually.

This topic applies to the MFC ODBC classes.

This topic explains:

- Your role and your options in selecting records.
- How a recordset constructs its SQL statement and selects records.
- What you can do to customize the selection.

Recordsets select records from a data source through an ODBC driver by sending SQL statements to the driver. The SQL sent depends on how you design and open your recordset class.

Your Options in Selecting Records

The following table shows your options in selecting records.

How and When You Can Affect a Recordset

Expand table

When you	You can

When you	You can
Declare your recordset class with the **Add Class** wizard	Specify which table to select from. Specify which columns to include. See Adding an MFC ODBC Consumer.
Complete your recordset class implementation	Override member functions such as `OnSetOptions` (advanced) to set application-specific options or to change defaults. Specify parameter data members if you want a parameterized recordset.
Construct a recordset object (before you call `Open`)	Specify a search condition (possibly compound) for use in a **WHERE** clause that filters the records. See Recordset: Filtering Records (ODBC). Specify a sort order for use in an **ORDER BY** clause that sorts the records. See Recordset: Sorting Records (ODBC). Specify parameter values for any parameters you added to the class. See Recordset: Parameterizing a Recordset (ODBC).

|Run the recordset's query by calling `Open` |Specify a custom SQL string to replace the default SQL string set up by the wizard. See CRecordset::Open in the *Class Library Reference* and SQL: Customizing Your Recordset's SQL Statement (ODBC).|

|Call `Requery` to requery the recordset with the latest values on the data source|Specify new parameters, filter, or sort. See Recordset: Requerying a Recordset (ODBC).|

How a Recordset Constructs Its SQL Statement

When you call a recordset object's Open member function, `Open` constructs a SQL statement using some or all of the following ingredients:

- The *lpszSQL* parameter passed to `Open`. If not NULL, this parameter specifies a custom SQL string or part of one. The framework parses the string. If the string is a SQL **SELECT** statement or an ODBC **CALL** statement, the framework uses the string as the recordset's SQL statement. If the string does not begin with "SELECT" or "{CALL", the framework uses what is supplied to construct a SQL **FROM** clause.
- The string returned by GetDefaultSQL. By default, this is the name of the table you

specified for the recordset in the wizard, but you can change what the function returns. The framework calls `GetDefaultSQL` — if the string does not begin with "SELECT" or "{CALL", it is assumed to be a table name, which is used to construct a SQL string.

- The field data members of the recordset, which are to be bound to specific columns of the table. The framework binds record columns to the addresses of these members, using them as buffers. The framework determines the correlation of field data members to table columns from the RFX or Bulk RFX function calls in the recordset's DoFieldExchange or DoBulkFieldExchange member function.
- The filter for the recordset, if any, contained in the m_strFilter data member. The framework uses this string to construct a SQL **WHERE** clause.
- The sort order for the recordset, if any, contained in the m_strSort data member. The framework uses this string to construct a SQL **ORDER BY** clause.

  Tip

  To use the SQL **GROUP BY** clause (and possibly the **HAVING** clause), append the clauses to the end of your filter string.

- The values of any parameter data members you specify for the class. You set parameter values just before you call `Open` or `Requery`. The framework binds the parameter values to "?" placeholders in the SQL string. At compile time, you specify the string with placeholders. At run time, the framework fills in the details based on the parameter values you pass.

`Open` constructs a SQL **SELECT** statement from these ingredients. See Customizing the Selection for details about how the framework uses the ingredients.

After constructing the statement, `Open` sends the SQL to the ODBC Driver Manager (and the ODBC Cursor Library if it is in memory), which sends it on to the ODBC driver for the specific DBMS. The driver communicates with the DBMS to carry out the selection on the data source and fetches the first record. The framework loads the record into the field data members of the recordset.

You can use a combination of these techniques to open tables and to construct a query based on a join of multiple tables. With additional customization, you can call predefined queries (stored procedures), select table columns not known at design time and bind them to recordset fields or you can perform most other data-access tasks. Tasks you cannot accomplish by customizing recordsets can still be accomplished by calling ODBC API functions or directly executing SQL statements with CDatabase::ExecuteSQL.

Customizing the Selection

Besides supplying a filter, a sort order, or parameters, you can take the following actions to customize your recordset's selection:

- Pass a custom SQL string in *lpszSQL* when you call Open for the recordset. Anything you pass in *lpsqSQL* takes precedence over what the GetDefaultSQL member function returns.

  For more information, see SQL: Customizing Your Recordset's SQL Statement (ODBC), which describes the types of SQL statements (or partial statements) you can pass to `Open` and what the framework does with them.

  > **Note**
  >
  > If the custom string you pass does not begin with "SELECT" or "{CALL", MFC assumes it contains a table name. This also applies to the next bulleted item.

- Alter the string that the wizard writes in your recordset's `GetDefaultSQL` member function. Edit the function's code to change what it returns. By default, the wizard writes a `GetDefaultSQL` function that returns a single table name.

  You can have `GetDefaultSQL` return any of the items that you can pass in the *lpszSQL* parameter to `Open`. If you do not pass a custom SQL string in *lpszSQL*, the framework uses the string that `GetDefaultSQL` returns. At a minimum, `GetDefaultSQL` must return a single table name. But you can have it return multiple table names, a full **SELECT** statement, an ODBC **CALL** statement, and so on. For a list of what you can pass to *lpszSQL* — or have `GetDefaultSQL` return — see SQL: Customizing Your Recordset's SQL Statement (ODBC).

  If you are performing a join of two or more tables, rewrite `GetDefaultSQL` to customize the table list used in the SQL **FROM** clause. For more information, see Recordset: Performing a Join (ODBC).

- Manually bind additional field data members, perhaps based on information you obtain about the schema of your data source at run time. You add field data members to the recordset class, RFX or Bulk RFX function calls for them to the DoFieldExchange or DoBulkFieldExchange member function, and initializations of the data members in the class constructor. For more information, see Recordset: Dynamically Binding Data Columns (ODBC).
- Override recordset member functions, such as `OnSetOptions`, to set application-specific options or to override defaults.

If you want to base the recordset on a complex SQL statement, you need to use some combination of these customization techniques. For example, perhaps you want to use SQL clauses and keywords not directly supported by recordsets or perhaps you are joining multiple tables.

See also

Recordset (ODBC)
Recordset: How Recordsets Update Records (ODBC)
ODBC Basics
SQL
Recordset: Locking Records (ODBC)

Recordset: How Recordsets Update Records (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

Besides their ability to select records from a data source, recordsets can (optionally) update or delete the selected records or add new records. Three factors determine a recordset's updateability: whether the connected data source is updateable, the options you specify when you create a recordset object, and the SQL that is created.

Note

The SQL on which your `CRecordset` object is based can affect your recordset's updateability. For example, if your SQL contains a join or a **GROUP BY** clause, MFC sets the updateability to FALSE.

Note

This topic applies to objects derived from `CRecordset` in which bulk row fetching has not been implemented. If you are using bulk row fetching, see **Recordset: Fetching Records in Bulk (ODBC)**.

This topic explains:

- Your role in recordset updating and what the framework does for you.
- The recordset as an edit buffer and the differences between dynasets and snapshots.

Recordset: How AddNew, Edit, and Delete Work (ODBC) describes the actions of these functions from the point of view of the recordset.

Recordset: More About Updates (ODBC) completes the recordset update story by explaining how transactions affect updates, how closing a recordset or scrolling affects updates in progress, and how your updates interact with the updates of other users.

Your Role in Recordset Updating

The following table shows your role in using recordsets to add, edit, or delete records, along with what the framework does for you.

Recordset Updating: You and the Framework

Expand table

You	The framework
Determine whether the data source is updateable (or appendable).	Supplies CDatabase member functions for testing the data source's updateability or appendability.
Open an updatable recordset (of any type).	
Determine whether the recordset is updatable by calling `CRecordset` update functions such as `CanUpdate` or `CanAppend`.	
Call recordset member functions to add, edit, and delete records.	Manages the mechanics of exchanging data between your recordset object and the data source.
Optionally, use transactions to control the update process.	Supplies `CDatabase` member functions to support transactions.

For more information about transactions, see Transaction (ODBC).

The Edit Buffer

Taken collectively, the field data members of a recordset serve as an edit buffer that contains one record — the current record. Update operations use this buffer to operate on the current record.

- When you add a record, the edit buffer is used to build a new record. When you finish adding the record, the record that was previously current becomes current again.
- When you update (edit) a record, the edit buffer is used to set the field data

members of the recordset to new values. When you finish updating, the updated record is still current.

When you call AddNew or Edit, the current record is stored so it can be restored later as needed. When you call Delete, the current record is not stored but is marked as deleted and you must scroll to another record.

> **Note**
>
> The edit buffer plays no role in record deletion. When you delete the current record, the record is marked as deleted, and the recordset is "not on a record" until you scroll to a different record.

Dynasets and Snapshots

Dynasets refresh a record's contents as you scroll to the record. Snapshots are static representations of the records, so a record's contents are not refreshed unless you call Requery. To use all the functionality of dynasets, you must be working with an ODBC driver that conforms to the correct level of ODBC API support. For more information, see ODBC and Dynaset.

See also

Recordset (ODBC)
Recordset: How AddNew, Edit, and Delete Work (ODBC)