Query By Image Content
QBIC™

Version 3.0

Programmer's Guide

IBM Corporation
Almaden Research Center
650 Harry Road
San Jose, CA 95120

Contents

Preface

This preface contains the following sections:

What is QBIC?

Query By Image Content (QBIC) is a technology that allows you to query collections of images by their content. "Query by content" means that you can query a collection of images in order to locate images that are similar to the query image, where similarity can be based on color, texture, or other image properties. For example, you can search for images that have predominantly red colors or striped textures, where the color and texture information is automatically computed. Queries by content complement traditional queries that use image file names or keyword descriptions.

QBIC does not provide semantic analysis, meaning for example, QBIC cannot locate your "favorite images," such as those that contain pictures of your cats, dogs, or other objects that require semantic understanding.

QBIC technology has been incorporated into some IBM software products, such as DB2 Image Extender and Digital Library.

This manual describes a set of C++ classes and software routines that implement QBIC technology. These routines were used to implement QBIC technology in the previously noted IBM products. If you want to know more about the technical details of QBIC, see the papers have we published and the references therein. The following are two recent papers:

- Niblack W. et al., "Updates to the QBIC System" in **Storage and Retrieval for Image and Video Databases** VI, Vol 3312, pp. 150-161, SPIE, January, 1998.
- Flickner M., et al., "Query by image and video content: The QBIC system", **IEEE Computer** 28(5), pp. 23-32, 1995.

This QBIC product includes:

- The libraries and header files that you need to implement QBIC functionality in your applications and to extend the QBIC feature extraction and search functionality
- Command-line executable modules and demo applications that are built on the QBIC API
- A “mini” web server used by the QBIC demo applications to demonstrate QBIC functionality

Intended Audience

This guide is intended for programmers who want to incorporate QBIC functionality into their applications and incorporate their own feature extraction capabilities into QBIC in order to extend QBIC functionality. It is assumed that users have a working knowledge of the C++ programming language.

Summary of Contents

This manual contains the following chapters and appendices:

Technical Support

For more information about QBIC, visit our web site at `http://wwwqbic.almaden.ibm.com`.

For technical information, contact:

IBM Corporation
Almaden Research Center
650 Harry Road
San Jose, CA 95120

Computer Science Department
attn: Mr. Xiaoming Zhu

408-927-1427

`xzhu@almaden.ibm.com`

Licensing Information

For licensing information, contact:

IBM Corporation
Almaden Research Center
650 Harry Road
San Jose, CA 95120

Business Development
attn: Mr.Ted Loewenberg

408-927-1202

`tedl@almaden.ibm.com`

Installing QBIC 1

This chapter contains the following sections:

When you install QBIC, you get:

- A set of command-line programs that can be used for database population and query based on a set of predefined QBIC features. You can build a complete application using these command-line programs.
- Libraries and header files for the QBIC API. You can use the API as an interface to QBIC directly from C and C++ code. You can even add new image feature extraction and query classes in order to extend QBIC functionality.
- A demo that allows you to evaluate QBIC functionality using a set of included images. In this demo, QBIC will start a mini web server. On NT and UNIX platforms, QBIC will also start a web browser to communicate with the web server for QBIC functionality.

NOTE: *You can also use the mini web server when developing and testing your own QBIC-based applications. However, when you deploy your application, it is recommended that you replace the mini web server with a full-scale server. See "Replacing the Mini Web Server" on page 181 for details.*

This chapter describes how to install the QBIC demo and classes. QBIC runs on the following platforms:

- AIX 4.1 and higher
- Linux 2.0.30
- Solaris 2.6
- Windows NT/95
- Macintosh OS8 on a PowerPC

Installation Steps

To install QBIC:

1 From the QBIC Development Kit (QDK), locate the correct QBIC distribution file for your platform:

Platform	Distribution file
AIX 4.1 or higher	qbaix_3.zip or qbaix_3.tgz
Linux 2.0.30	qblinux_3.zip or qblinux_3.tgz
Solaris 2.6	qbsun_3.zip or qbsun_3.tgz
Windows NT/95	qbnt_3.exe
Macintosh OS8	qbmac_3.sit

2 Locate or create a directory where you want to install QBIC. When the QBIC distribution file is uncompressed, it creates a `qbic` directory. For example, if you uncompress QBIC files to the `/usr/local` directory, QBIC will install in the `/usr/local/qbic` subdirectory.

NOTE: *Any references to a directory or path beginning with qbic is assumed to be relative to the installation directory (e.g. /usr/local). In this manual, we use the UNIX forward slash (/) for directory separators. For PC-based platforms, please substitute the backslash (\).*

3 Uncompress the QBIC distribution file in the directory where you want to install QBIC.

- For UNIX platforms, type:

```
unzip source_file.zip
```

or

```
gunzip -d source_file.tgz
tar -xvf source_file.tar
```

NOTE: *You need the unzip or gunzip utility to uncompress the QBIC distribution file. These utilities can be found on the Internet.*

- For Windows NT or Windows 95 platforms, run the self-extracting program `qbnt_3.exe`, and then specify the directory in the pop-up window.
- For Macintosh, double-click the self-extracting program `qbmac_3.sit`, which will create a qbic folder called qbmac_3.

Starting the Demo

If you want to run the QBIC demo, go to the `qbic` directory and type:

```
qbicdemo
```

NOTE: *On the Macintosh, you must start the demo manually:*

1. Start the qbictcl program in the `qbic/bin` *directory.*
2. In the command shell of qbictcl type: `source ../script/qbserver.tcl`
3. Start Netscape or Internet Explorer.
4. Enter the following URL: `http://MachineName:3456`*, where* `MachineName` *is your Macintosh's Internet domain name. If you are unsure, you can enter* `http://127.0.0.1:3456`.

See Chapter 5 for detailed information about running QBIC demos.

QBIC Installation Directories

The QBIC API and related files are installed to the following directories:

Directory	Description
qbic/QbicApi	Source, sample programs, libraries, header files
qbic/bin	Executables, dlls (PC only), scripts
qbic/classes	Java applets used for the web demo
qbic/docs	PDF and HTML on-line documentation
qbic/html/gifs	Images for the web demo
qbic/html/images	Image directory
qbic/html/text	Text files containing image descriptions or keywords
qbic/html/thumb	Thumbnail directory
qbic/QbicData	QBIC database for catalog files
qbic/script	tcl script for a QBIC mini web server

About QBIC 2

Query By Image Content (QBIC) is a technology that allows you to query collections of images by their content. “Query by content” means that you can query a collection of images in order to locate images that are similar to the query image, where similarity can be based on color, texture, or other image properties. For example, you can search for images that have predominantly red colors or striped textures, where the color and texture information is automatically computed. Queries by content complement traditional queries that use image file names or keyword descriptions.

This chapter provides a brief overview of QBIC including:

What You Get with QBIC

When you install QBIC, you get:

- The libraries and header files that you need to implement QBIC functionality in your applications and to extend the QBIC feature extraction and search functionality
- Command-line executable modules and demo applications that are built on the QBIC API
- A “mini” web server used by the QBIC demo applications to demonstrate QBIC functionality

Image Types Supported by QBIC

QBIC supports the following commonly used image formats. QBIC recognizes image format based on the image file extension:

- OS/2 and Windows bitmap image, BA-type only (bmp)
- GIF 87 or 89a (gif)

- JPEG (jpg/jpeg)
- PBMPLUS: portable graymap (pgm)
- PBMPLUS: portable pixmap (ppm)
- TIFF 5.0 image, including CCITT Group 3 and Group 4 fax image (tiff/tif)
- Targa (tga)

In addition to the standard image formats listed above, the QbGenericImageDataClass, described on page 125, can also read a QBIC Picker Image Description string either in memory (ReadPickerImageDescriptionString) or in a file (ReadPickerImageDescriptionFile). This description string is a simple encoding of an image in a text string. Currently, this method only recognizes rectangles. The string format is:

```
Dwidth,height:Rulx,uly,rwidth,rheight,R,G,B:...
```

where:

- D specifies the Drawing area with the dimensions width and height. The origin is in the upper left corner.
- : (colon) is the field separator
- R specifies a Rectangle with the upper left corner at ulx,uly, relative to the drawing area's origin. Dimensions are rwidth, rheight. Color is specified in RGB format as R,G,B.

You can specify multiple rectangles (R strings), but the first rectangle must be "painted" in the drawing area first, and subsequent rectangles which overlap are painted over it. QBIC considers any area that is not painted is "to be ignored". Color information inside the "to be ignored" region does not affect the QBIC distance evaluation.

The QBIC Picker Image Description string is useful for sending image descriptions from a client to a server so that the server can construct a query image based on the description, and then search for similar images in the database. The description string is heavily used in the QBIC demo.

Overview of QBIC's Image Query Processes

The API computes, stores, and retrieves data in databases and catalogs.

- A **database** is a named collection of data. It may be a DB2 or Oracle database, a collection of dbm files (dbm is a set of UNIX utilities that provide database-like functions), or any other DBMS.
- A **catalog** is a named set of tables within a database.

For example, a user may create the MY_IMAGE_DATA database with two catalogs, VACATION_PICTURES and FABRIC_SAMPLES.

In the case of DB2 or Oracle, this will create the MY_IMAGE_DATABASE database containing one set of tables named VACATION_PICTURES and another set of tables named FABRIC_SAMPLES.

In the case of dbm, this will create a directory named MY_IMAGE_DATABASE and two sets of files in that directory. One set will have file names with the extension VACATION_PICTURES, and the other with the extension FABRIC_SAMPLES.

Phases of a Query

Querying by content requires two phases:

- **Database creation**: A preprocessing step to compute numeric features (also called metadata) for an image or set of images, and store them in a database.
- **Database query**: The run-time step that uses the computed features to find images similar to a query specification.

During the database creation phase, you use one or more feature classes to compute the features of input images as numeric values. Each feature class creates a feature table in the database where these computed values are stored.

During the database query phase, QBIC compares feature data in the query with the computed data in the feature tables. A query can search on one or more features for similarity.

Types of Query

A simple query involves only one feature. An example of a simple query would be to find images in the database that have a color distribution similar to the query image.

A complex query involves more than one feature and can be either a multi-feature or a multi-pass query. In a multi-feature query, QBIC searches through different types of feature data in the database in order to find images that closely resemble the query image. All feature classes are treated equally during the database search, and all involved feature tables are searched at the same time. An example of a multi-feature query would be finding images in the database that have a color distribution *and* texture similar to a query image.

A “pass” is a single feature or multi-feature search. In a multi-pass query, the output of an initial search is used as the input for the next search. QBIC reorders the search results from a previous pass based on the “feature distances” in the current pass. An example of a multi-pass query would be finding images in the database that have a color distribution similar to the query image, and then reordering the results based on color composition.

For multi-feature and multi-pass queries, you can weight features to specify their relative importance, which provides flexibility for advanced applications where the returned results must be fine tuned.

QBIC queries can be classified into the following types:

- Query By Example: This query type is used most often. In this type of query, you locate images in a database that are similar to the “example” query image (whose feature data has already been computed by QBIC).
- Query By Image: If you have an image whose feature data has not yet been computed by QBIC, and you want to find similar images in the database, you invoke the Query By Image type of query. QBIC computes the feature data for the input image, and then compares it with data in database.
- Query By Picker: This is a custom query where you “paint” an image or part of an image, and use QBIC to find similar images inside database. In this type of the query, you do not have to paint every part of image–only the part that contains the features you are looking for. The “unpainted” areas are treated as “to be ignored” and will not affect query results.

The Predefined QBIC Feature Class

QBIC has predefined methods to compute image properties. Each method is implemented as a C++ class.

- The QbColorFeatureClass method computes the average RGB colors for images. Image similarity is based on these three average color values.
- The QbColorHistogramFeatureClass method computes the color distribution for each image in a predetermined 256 color space. Image similarity is based on the similarity of the color distribution.
- The QbDrawFeatureClass method computes the dominant color in each of a set of predetermined regions of the input image. Image similarity is based on the color similarity for each region so that the measure is also sensitive to position color.
- The QbTextureFeatureClass method computes texture information. Image similarity is based on several texture attributes such as directionality, coarseness, and contrast.
- The QbTextFeatureClass indexes any text information associated with each image, and allows you to use keywords to narrow the QBIC search.

Error Checking and Handling

The QBIC API implements extensive error checking routines that keep track of internal errors. If any QBIC method call returns a nonzero return code, it implies that an error occurred. If an error occurs in the class constructor (or any other method that does not return a value), you can query the internal state of the instance of the class by using the IsOk() method. If this method returns False, it means that the instance has failed, most likely due to a memory allocation failure.

QBIC includes error handling routines, which you can call at the beginning of your program or before QBIC calls. See “QBIC Error Handling Routines” on page 165 for information.

Advanced QBIC Features Available by Special License Agreement

The QBIC API contains advanced features that can speed up a QBIC search and improve QBIC feature extraction. These advanced features are not supported in this QBIC release, although some methods that support these advanced features are included in this API. If you are interested in implementing any of the following features in your applications, contact IBM. See “Licensing Information” on page xiii for how to contact IBM.

SubParts

Many of the QBIC feature classes support the concept of SubPart or Object features. SubPart and Object features represent feature data that is computed for only part of an image, the “part” being selected by an mask image. Queries are computed for these subimages using the SubPartFeatureClass.

The mask must:

- Have the same dimensions as the image itself
- Be bi-level (although not necessarily binary). The background, which represents the area you want to ignore, must be black or have a zero (0) pixel value. The foreground, which represents the part of the image for which features should be computed, must be white or have a non-zero pixel value.

Cluster Indexing

To speed up queries for very large databases, QBIC has a method that organizes feature vectors into clusters for a given feature class. This feature is called cluster indexing.

If cluster indexing is enabled, QBIC will match against a subset of the feature vectors during a query, thereby reducing the query time. The cluster tables are stored in special database files. Modifications, such as the addition or deletion of records, are handled by the QbCatalogClass.

Precomputed Queries

This QBIC advanced feature provides the ability to store the top results of the image keys inside a database to any query keys also inside the database. Using this feature if a user submits a query using a key inside the database which is often the case, QBIC only needs to read a record from a special database file and return the result.

As with cluster indexing, modifications to the database are handled by the QbCatalogClass. Extra data is stored in the global information.

Overview of the QBIC API 3

The QBIC API has two levels of functionality, as described in the following sections:

QBIC Command-Line Programs

At the higher level are command-line programs such as QbMkDbs, QbQBE, QbMkThmb, and QbDumpDb. These programs have been compiled on different platforms, and provide the following functionality:

- Populating a QBIC database (QbMkDbs)
- Querying a QBIC database (QbQBE)
- Generating thumbnail images (QbMkThmb)
- Dumping the contents of feature data from dbm files (QbDumpDb)

You can run these programs in a command shell, or call them from your application using system or execl calls.

QBIC provides a high-level "wrapper" class, called QbicWrapClass, around the functions mentioned in QBIC Command-Line Programs, making the functions accessible directly from C++.

You simply create a QbicWrapClass object in your application, and then invoke the appropriate class method for the needed functionality. Details about the class and methods are described in "QbWrapClass" on page 31.

QBIC's Abstract and Fully Implemented Classes

At lower level, QBIC provides a set of abstract C++ classes and a few fully implemented classes to provide QBIC functionality, such as database connection and management, feature data computation, and so on. The QBIC abstract C++ classes are classes that use pure virtual functions to define the interface, forcing the derived classes to specify the implementation. Programs cannot create instances of abstract classes–only pointers to abstract classes are allowed.

The QBIC API is designed to be extensible, which means that you can add new feature computation classes to QBIC in order to extend its functionality. Although the

API was designed primarily for query by image content, it is general enough to process other media data types, such as video, audio, and text.

If you are familiar with the Object Management Group's Object Query Service (OQS) model, the low-level QBIC API maps classes and methods to the OQS model when possible, but it is not a full OQS implementation. For example, many of the QBIC classes implement the ToQueryString and FromQueryString methods to convert an instance of the class to an ASCII string. You can pass the ASCII string across process or machine boundaries in a client/server environment.

Abstract class definitions are briefly described in the following sections:

- "QbDatumClass" on page 13
- "QbGenericDataClass" on page 13 (Generic data to describe image, text, and audio data for feature computation/indexing)
- "QbConnectClass" on page 14 (Connection to database/filesystem)
- "QbDatabaseClass" on page 14 (Database for storing feature/indexing data)
- "QbDbIteratorClass" on page 15 (DatabaseIterator to iterate over a database)
- "QbKeyDatabaseClass" on page 15 (Database for returning query results and storing key search lists)
- "QbFeatureClass" on page 16 (Features to describe the extracted information used for indexing/search)
- QbParameterClass, described with "QbGenericDataClass" on page 13 (Parameters to allow for the specification of various query and population related parameters)

For details, please see later chapters in this manual. Header files that provide details of the C++ definitions for these classes are available in the `qbic/QbicApi/include` subdirectory. Using this framework, you can define generic functions to populate and query a database.

The low-level class hierarchy is shown in Figure 1 on page 13.

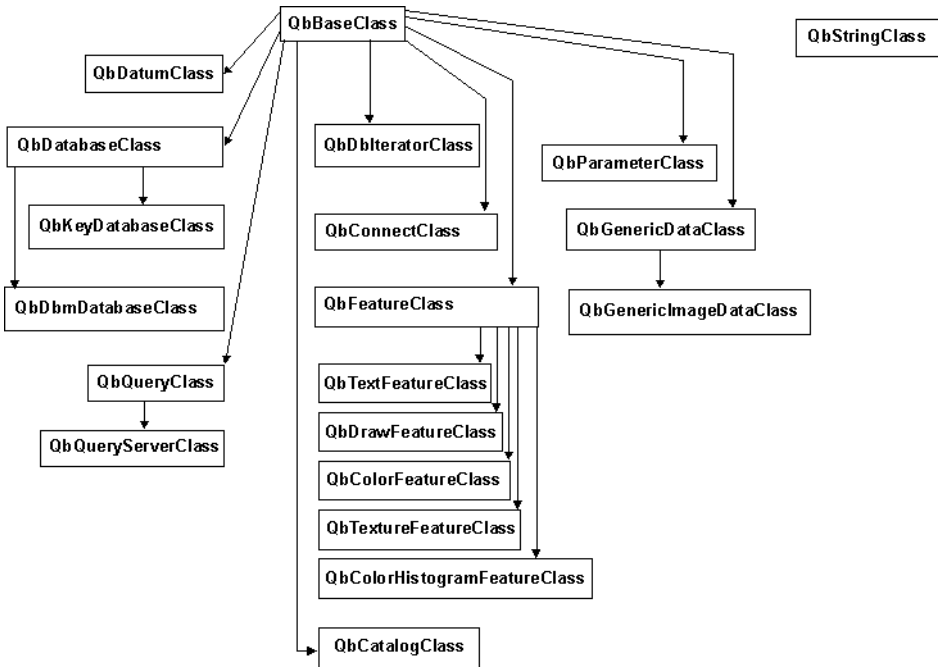

Figure 1. Low-Level Class Hierarchy

The hierarchy uses only single inheritance. The naming convention for classes is Qb*DescriptivePhrase*Class. Member functions are named in mixed case with the first character in uppercase. Variables are named in mixed case with the first letter in lowercase. For example, QbThisIsAClass, ThisIsAMemberFunction and thisIsAVariable are consistent with the naming convention.

QbDatumClass

QbDatumClass contains a pointer to a byte string, the length of the byte string, and a character description of the format. The member functions Get and Set allow getting and setting of the pointer and size variables. The functions GetFormatInfo and SetFormatInfo allow access to the format information. QbDatumClass supports marking of allocated memory so it can be deleted, freed, or left alone in the destructor or when reassigned using Set. It also has an assignment operator.

QbGenericDataClass

The class QbGenericDataClass is a class that defines generic data, such as image, video, character, and audio. This class encapsulates the data between the feature extraction process, the user interface, and external data, such as image files.

QbParameterClass is a class to encapsulate the parameters associated with a query. Both classes define the member functions ToQueryString and FromQueryString. These member functions are used to convert the class to an ASCII query string to allow the query to be passed from the client to the server query engine.

QbParameterClass is a full implementation of an array of name/value pairs for any of the feature classes.

QbGenericDataClass is a full implementation that gives a standardized way to pass query by example or query from a sampler between client and server. It also gives a standard way of passing domain-specific information between client and server. It does nothing to document the format of the domain-specific data. To do this, the class should be subclassed, and the domain-specific data can be specified. In the subclassed version, ToQueryString should take the domain-specific data and put it into QbGenericDataClass' QbDatumClass field named "data." Then it should call the base class ToQueryString function to encode both the generic information and the domain-specific information. FromQueryString reverses this process. An example of this process is in QbGenericImageDataClass.

QbConnectClass

To obtain a connection to a database or a filesystem, use QbConnectClass. The Connect/Disconnect call establishes a connection to a database or mounts a filesystem–processing that is expensive and typically only done once. The function QbGetConnection in the file `QbCreate.cpp` implements the simple one-connection-per-address-space connection needed by the current engine.

QbGetConnection takes two arguments: the name of the data set, and the name of the connection class. It returns a const pointer instance of the connection class. Because it is const, you cannot Disconnect, Connect or delete the pointer. However, you can create instances of a database appropriate for this connection using the function CreateDatabaseClass and CreateKeyDatabaseClass.

Use the function QbDestroyConnection, also in `QbCreate.cpp`, when you are finished with the connection. The current implementation for dbm files creates a directory using the passed name of the database.

QbDatabaseClass

To store features, the class QbDatabaseClass defines the abstract interface to a database. The virtual abstract member functions Open, Close, Insert, Update, Retrieve, Delete, Commit, and CreateIterator need to be defined by the specific implementation, such as DB2, Oracle, or dbm. Open and Close open and close a namespace for subsequent operations. If the namespace does not exist Open creates it. Open also takes an optional second argument that specifies the mode, either "w" for write, or "r" for read. The default is to open in the mode supported by the connection to the database. However, a call to Open in "w" mode with a "r" connection will fail.

To remove the namespace use the CloseAndRemove member function. The Insert member function inserts the key/value pair into the database and fails if the key already exists. To update an existing record, use the Update function. Update fails if the key does not exist. Given a key, Retrieve returns the value and Delete deletes the record. Because the database object often keeps state, copying of QbDatabaseClass is not allowed. If multiple objects are needed, the program is free to create them.

CreateIterator is an abstract virtual function that creates an instance of a database iterator.

QbDbIteratorClass

To define the interface to lists, use QbDbIteratorClass. This class has only three virtual functions: More, Next, and Reset (similar to the OQS iterator functions). More returns true if there is more data. Next returns the next record. Reset resets the iterator so that a future call to Next returns the first record.

To create an instance of QbDbIteratorClass, use the QbDatabaseClass member function CreateIterator. This assures the iterator is consistent with the database. The state of an iterator is undefined if an Insert, Delete, or Update is done to the database after the iterator was created. Once done using the iterator, do not forget to delete it (in order to avoid memory leaks).

QbKeyDatabaseClass

To return the results of queries and to specify restrictions list of keys, use QbKeyDatabaseClass. A restriction list is a list of keys used to restrict a query. A common example is in a query such as “Find images by artist Picasso with this shade of red.” A text query on “Picasso” returns a set of keys of images. These keys are then passed to the query function as a restriction list, together with the content-based query specification for “red,” to obtain the final, ordered query results.

QbKeyDatabaseClass is a database of keys with distance values. Iterations on this database are required to return items in increasing distance order and to use key order in the event of a tie. QbKeyDatabaseClass is derived form QbDatabaseClass and includes several additional member functions.

IsRanked and SetRanked allow the user to query and set the ranked state of a database. Even though the database is always sorted using distance, this sort may not be valid. For an example, see the FilterResult function in the QbFeatureClass. GetDistance and SetDistance return and set the distance of a value datum.

GetNumberOfKeys returns the number of keys (records) in this database. The return results of a query are always ranked. When a QbKeyDatabaseClass is used to define a restriction list, use the same distance for all elements, or call SetDistance with no distance argument (because it defaults to use a distance of -1.0). The function RetrieveMax is a fast path to retrieve the element with the largest distance in the list. An efficient implementation of this function enhances query engine performance.

Finally, to pass restriction lists to a query, the member functions ToQueryString must be used. This process can be reversed using the function FromQueryString. ToQueryString creates a blank terminated list of key distance pairs, and FromQueryString reinserts these into a QbKeyDatabaseClass object.

QbFeatureClass

QbFeatureClass is used to represent a feature. The member functions ComputeFeatures, Distance, ExecutionCostForDistance, FilterResult, and ExecutationCostForFilter are required virtual functions. ComputeFeature takes two arguments, a pointer to generic data and a query specification. Distance returns the distance between this instance of a feature class and another instance of the feature class. The FilterResult function arguments are a connected database, an optional key sublist of this database, the number of results requested by the query, the name of the catalog, and the query parameters. FilterResults can return a ranked or an unranked list. If the list is not ranked, the returned distances do not reflect final query ordering and the caller should use the Distance function to determine the final ordering. The functions GenericDataClassname and ParameterClassname return the name of the QbGenericDataClass and QbParameterClass class needed by this feature.

GetDefaultParameters returns a pointer to the default parameters for this feature class. This default parameter data is normally implemented as class static member data. The return type is const so that the caller cannot delete the result. Any implementation where the caller does not delete the results is allowed. To change the default parameters, make a copy of the default parameters using the assignment operator before changing the values.

The member function UpdateGlobalFeatureInfo allows the implementor of a feature to keep various statistics such as sum and sum of squares, which are needed at distance computation time. This allows the implementation Mahalanobis distance, if required. The base class implementation of this implements a simple counter to count the number of records in a database. You can use the parameter information to determine when to use global information. This is done using the class function ParameterUpdateFromGlobalInfo. Currently, only the Texture feature uses a variance weighted distance. In this case, the sum and sum of squares are stored as the global information for the feature. At query time, a parameter value of V*xx*, where *xx* is an optional numerical weight, and V stands for variance, implements an inverse variance weighting. That is, a given feature is weighted by a coefficient given as 1/V*xx* for all data in the collection. For example, V5 means (5/variance) weighting. At query time, the global information is read from the catalog and used to update the weights that will be used in the distance computation. Note that the query engine has no knowledge of the type of weighting or how the distance will be computed. This is left entirely up to the feature designer.

Features need to be stored in a database. This is accomplished using the member functions ToByteString and FromByteString. These functions convert the contents of the class into a byte string, and set the contents of the class from a byte string. The argument of ToByteString or FromByteString is a reference to the QbDatumClass.

Integrating the QBIC API Into Your Application

4

This chapter contains the following regarding the integration of the QBIC API into your applications:

QBIC API Levels of Complexity

When you integrate the QBIC API with your application, you can choose between the following levels of complexity:

- At a high level, you can use the command-line programs such as QbMkDbs (database population), QbQBE (database query), and QbMkThmb (thumbnail generation) to run QBIC functions. Or, by using a scripting language such as tcl or perl, you can build a complete application by invoking these functions, capturing their output (written to stdout), and building a suitable GUI.
- You can include the QbicWrapClass in your C++ code and call its methods, such as QbicWrapDBS, QbicWrapQBE, and QbicWrapThm for the corresponding functionality described in the previous bullet. QbicWrapClass provides a "wrapper" around the functions QbMkDbs, QbQBE, and QbMkThmb, making them accessible directly from C++.
- If you want lower-level access to QBIC functions (for example, to store QBIC features in your own database, or to add new image analysis features), you can use the lower-level QBIC API described in later chapters.

Integrating the higher-level QBIC functionality is simple and straightforward, and you should use this approach whenever possible. The only drawback of this approach is that you have to use QBIC's default database implementation (dbm) to manage content.

If you want to use your own DBM system to store QBIC feature data, you must use the lower-level QBIC API. A set of sample programs, described in the following section, will help to get you started.

Sample Programs

QBIC includes a collection of sample programs with source code that demonstrate how to incorporate QBIC functionality into your applications. The executables for more complete programs, including QbMkDbs and QbQBE, QbMkThmb, and QbDumpDb are in the `qbic/bin` subdirectory. These programs are close to what you would need in a real application. Program descriptions and instructions on how to use them are given in “Running QBIC Demo Programs” on page 21.

The source code for these programs is located in the `qbic/QbicApi` subdirectory. You will notice after reading the source code that each of these programs simply creates a QbicWrapClass object inside the program, and then invokes the object’s method such as QbicWrapDBS, QbicWrapQBE, QbicWrapThm, and QbicWrapDumpDb for the needed functionality.

A set of programs using the lower-level QBIC API is also included. Each of these programs is designed with limited functionality for the sake of simplicity. They are located in `qbic/QbicApi/examples` subdirectory. Get familiar with these low-level sample programs before reading the high-level implementation, like QbicWrapClass:

- “QbGetImage.cpp” on page 18
- “QbImgDis.cpp” on page 18
- “QbKeyDis.cpp” on page 19
- “QbKeyIte.cpp” on page 19

QbGetImage.cpp

QbGetImage.cpp demonstrates how to encapsulate image data for feature extraction using the QbGenericImageDataClass, which is derived from QbGenericDataClass.

This program reads an input image and prints image information, including size and RGB value at a specified pixel position.

NOTE: *QbGenericImageDataClass has its own color quantization routine that reduces the number of image colors based on QBIC’s own color palette.*

QbImgDis.cpp

QbImgDis.cpp demonstrates how to compute the distance between two input images using the QbColorHistogramFeatureClass feature. The program shows how to use QbFeatureClass to compute image feature data.

This program creates QbColorHistogramFeatureClass objects, reads the image, and extracts feature data based on input image data. Once the feature data for two images has been extracted, the image distance is computed.

QbKeyDis.cpp

QbKeyDis.cpp demonstrates how to compute the distance between two input keys using the QbColorHistogramFeatureClass feature. In this example (as compared to the previous one), the image features have already been computed and are stored in a database.

This program retrieves the feature data for distance computation by connecting to a database. It also shows how to use the QbGenericImageDataClass to store computed feature data and raw image data.

QbKeyIte.cpp

QbKeyIte.cpp demonstrates how to use the QbDbIteratorClass to compute the first 10 image key distances from the query key.

NOTE: *The results are not ordered based on the distance from the query image. To do that, you must use the QbKeyDatabaseClass.*

This program also shows how to use the QBIC error handling routine to print any error messages generated by the API.

Deriving Your Own Feature Extraction and Matching Classes

To help you derive your own feature extraction classes, we provide a sample class `QbExampl.hpp` and `QbExampl.cpp` located in the `qbic/QbicApi` directory. These modules implement a skeleton class derived from QbFeatureClass. It is recommended that you read the extensive comments included in the module code.

In general, writing and including an additional feature class involves the following steps:

1. Subclass QbFeatureClass with your new feature class.

NOTE: *Important! The substring formed by the first six characters after a leading "Qb" (if present) must be unique among all feature classes. This prevents file naming problems in the catalog, which uses the 8.3 file naming convention. (File name is eight characters long; file extension is three characters long)*

2. Implement the required virtual functions, as they apply to the feature.
3. Add the appropriate clause to QbCreateFeature in `QbCreate.cpp`.
4. Recompile `QbCreate.cpp` and the new class.
5. Relink the QBIC API library and the main programs, including QbMkDbs and QbQBE. A sample makefile for supported platforms is provided in the `qbic/QbicApi` subdirectory.

Running QBIC Demo Programs

5

This chapter contains the information you need to run the web-based QBIC demo, run the QBIC command-line demo programs, set demo program environment variables, and add your own features classes to use with the QBIC demos, as described in the following sections:

Running the Web-Based QBIC Demo

NOTE: *You need Netscape Navigator version 3.01 or later to run the web-based QBIC demo.*

To start the web-based QBIC demo:

1. Go to the `qbic` directory.
2. Run `qbicdemo`.

NOTE: *On the Macintosh, you must start the demo manually:*

1. Start the qbictcl program in the `qbic/bin` *directory.*
2. In the command shell of qbictcl type: `source ../script/qbserver.tcl`
3. Start Netscape or Internet Explorer.
4. Enter the following URL: `http://MachineName:3456`*, where* `MachineName` *is your Macintosh's Internet domain name. If you are unsure, you can enter* `http://127.0.0.1:3456.`

Running QBIC Command-Line Programs

QBIC furnishes a collection of programs that provide QBIC functionality via the command-line. You can implement these programs in your own applications, or you can use them as examples when designing and writing your own programs:

- “Building a Catalog with QbMkDbs” on page 22
- “Querying the QBIC Catalog Using QbQBE” on page 24
- “Generating Thumbnails Using QbMkThmb” on page 26
- “Dumping Databases Using QbDumpDb” on page 27
- “Extending tcl script with QBIC Functionality Using qbictcl” on page 27

The QBIC database implementation is derived from the Berkeley dbm system. Therefore:

- To connect to a database means to create a directory
- To create a catalog you must generate a set of files in the directory with the same catalog name as the file extension

All of these programs expect command line input. If input is not provided, or if the input is incorrect, the program prints a summary of the command line syntax on the screen so you can read the required input and proper syntax.

You can set environment variables for these programs, as described in “Setting QBIC Environment Variables” on page 28.

Source code for these programs is in the `qbic/QbicApi` subdirectory.

Building a Catalog with QbMkDbs

QbMkDbs builds a catalog. (That is, it computes features describing image content properties, and adds them to the feature tables in the catalog.) It also has functions that can add an item to a sampler, update or delete an item from a feature table or sampler, and remove an entire catalog.

NOTE: *Removing an entire catalog can be dangerous. This is because you can create several catalogs in the same database, resulting in several sets of files that end with the catalog name as the file extension.*

Command Line Input and Environment Variables

QbMkDbs expects a set of command line inputs:

- Database name
- Catalog name
- QBIC feature name(s)
- Input image names or input file name (in which image names are listed)

QBIC uses the input image names as the key to feature data. If you decide that the key should not contain path information, set the QbicImagePath environment variable to point to the image directory, or start QbMkDbs in the directory where images are stored because QBIC always searches the current directory for a given input file name.

If you want to include path information as part of the key, set the QbicImagePath environment variable to point to the parent directory name appended with the key name that corresponds to the path for the input image, or start QbMkDbs in that directory. For example, if you have images in `/qbic/images/ibm33/flower01.jpg`, and you want the key to be `ibm33/flower01.jpg`, then you should set the environment variable as `QbicImagePath=qbic/images` or go to `qbic/images` to start QbMkDbs.

The database name and catalog name can be passed to QbMkDbs using -d and -c options. If a database name is not passed on the command line, QBIC will use the database name provided by the environment variable QbicDatabaseName. If QbicDatabaseName is not set, QBIC will default to QbicData. If the catalog name is not passed on the command line, QBIC will default to test.

QBIC feature names can be passed to QbMkDbs using a set of -f options. Each option describes a feature that QBIC must have implemented in order to extract data from the input image.

There are two basic operations that this program performs. First, it adds feature tables to the catalog. Second, it adds records to each of the feature tables (and modifies global information) for each image name specified. Though these two operations can be combined in one step, we recommend separating these two functions.

Example

For example, assume the following case:

1. Under `image/ibm33` there are images named `flower01.jpg flower02.jpg`, and so on.
2. Under `text/ibm33` the corresponding keyword information is in `flower01.jpg.txt`, `flower02.jpg.txt`, and so on.
3. A QBIC database named `/qbic/QbicData` and a catalog named `ibm33` need to be created.
4. QBIC keys should contain the image name, without any path information.
5. Features described by QbColorHistogramFeatureClass, QbTextureFeatureClass, and QbTextFeatureClass are needed.

The two-step method to create the QBIC database is go to `image/ibm33` and issue the following command:

```
QbMkDbs -c ibm -d /qbic/QbicData -f QbColorHistogramFeatureClass -f
    QbTextureFeatureClass -f QbTextFeatureClass
```

The result is that a directory named `/qbic/QbicData` is created, together with a set of dbm files:

- `/qbic/QbicData/FeatIdx.ibm33`
- `/qbic/QbicData/ColorHiF.ibm33`
- `/qbic/QbicData/TextureF.ibm33`
- `/qbic/QbicData/TextFeaF.ibm33`

To add images to the catalog for these features, use the following command:

```
QbMkDbs -c ibm33 -d /qbic/QbicData flower01.jpg flower02.jpg ...
```

or

```
QbMkDbs -c ibm33 -d /qbic/QbicData -l imageList
```

where imageList is a plain text file containing one image file name per line (with no blank lines).

QbMkDbs will report errors if it tries to add or insert a record to the catalog that already contains that record. Besides the insert mode, QbMkDbs has three modes for updating records. These are:

- Incremental update, which inserts missing feature data for an existing record (-i option)
- Update, which replaces all feature data for an existing record (-u option)
- Force update, which replaces existing and inserts missing feature data, even if the record is not currently in the catalog (-U option)

This program also supports deleting records and deleting the entire catalog. The command line syntax supports SubParts. Contact IBM for information about SubParts. (See “Licensing Information” on page xiii for contact information.)

Querying the QBIC Catalog Using QbQBE

QbQBE is designed to query the QBIC database and get other QBIC-related information. It can be used to query the database for images similar to the query image or key. It can also provide information such as which features were extracted for a particular catalog, or which features are enabled in the current QBIC release. It can even return the actual feature data for a query object.

Command Line Input and Environment Variables

QbQBE expects a set of command line inputs:

- Database name
- Catalog name
- QBIC feature name(s)
- Input image names or input file name (in which image names are listed)

Database query -d and -c options are needed to provide the program with database and catalog names. A set of -f options indicate which QBIC feature(s) to use in the query.

If more than one -f option is specified, QbQBE will perform a multi-feature or multi-pass query, depending on the input feature parameters.

If you do not supply input, environment variable settings are used.

Example

For example, assume the following case:

1. A QBIC database named `/qbic/QbicData` and a catalog named `ibm33` were created.
2. Image keys are `flower01.jpg flower02.jpg`, and so on.
3. QbColorHistogramFeatureClass, QbDrawFeatureClass, QbTextureFeatureClass, and QbTextFeatureClass features were extracted.

To find the top five images that are most similar to `flower02.jpg` in the `ibm33` catalog using the QbColorHistogramFeatureClass feature, use the following command:

```
QbQBE -c ibm33 -d /qbic/QbicData -f QbColorHistogramFeatureClass -i
    flower02.jpg -n 5
```

To find the default number of images (20) that are similar to `flower02.jpg` in the `ibm33` catalog using the QbColorHistogramFeatureClass and QbTextureFeatureClass features, with 70 percent weight on color histogram and 30 percent weight on texture in distance evaluation (multi-feature query), use the following command:

```
QbQBE -c ibm33 -d /qbic/QbicData -f QbColorHistogramFeatureClass:W=.7
    -f QbTextureFeatureClass:W=.3 -i flower02.jpg
```

To find the default number of images (20) that are similar to `flower02.jpg` in the `ibm33` catalog using the QbColorHistogramFeatureClass feature and to reorder the top returned results based on the QbDrawFeatureClass feature, with 50-percent weight on color histogram and 50-percent weight on the draw feature as the combined distance for reordering (multi-pass query), use the following command:

```
QbQBE -c ibm33 -d /qbic/QbicData -f
    QbColorHistogramFeatureClass:W=0.5:O=0 -f
    QbDrawFeatureClass:W=0.5:O=1 -i flower02.jpg
```

To dump the feature data for `flower02.jpg` using the QbColorHistogramFeatureClass feature, use the following command:

```
QbQBE -c ibm33 -d /qbic/QbicData -F QbColorHistogramFeatureClass -i
    flower02.jpg
```

To list all features that were extracted from `ibm33` catalog, use the following command:

```
QbQBE -c ibm33 -d /qbic/QbicData -l
```

To list all feature classes that the current QBIC release can extract from input images, use the following command:

```
QbQBE -L
```

To return random image keys from `ibm33` catalog in the `/qbic/QbicData` QBIC database, use the following command:

```
QbQBE -c ibm33 -d /qbic/QbicData -r
```

The -i key option in the command line can be replaced by the -I imageName option to run a query with the image imageName stored locally on the system (that is, one not already stored in the database). In this case, QbQBE computes the feature from the input image, and then does the query using the image.

Generating Thumbnails Using QbMkThmb

QbMkThmb creates thumbnail images for displaying query results. By default, the program will generate thumbnails using QBIC's own palette (four red, eight green, and four blue colors). The thumbnails are 100-by-100 pixels in both dimensions. The aspect ratio is maintained in the thumbnail. Because most images are not perfectly square, QbMkThmb pads any extra space with black pixels.

Command line options allow you to change the default behavior, as shown in the following examples.

Examples

To create thumbnails with an input image name and output thumbnail name:

```
QbMkThmb imageName thumbnailName
```

This command reads the image file named imageName from either the current directory or the search path specified by the environment variable QbicImagePath. It then generates an output file called thumbnailName in the current directory with the dimensions 100-by-100 pixels. Depending on the extension of the thumbnailName specified, the thumbnail can be in any of the supported QBIC image formats.

To generate multiple thumbnails:

```
QbMkThmb -l imageList
```

imageList is a file containing a list of files with one image name per line and no blank lines. (This can be the same file used for QbMkDbs.)

In this example, the generated thumbnails are placed in the current directory, have the dimensions 100-by-100 pixels, and are padded, if necessary. Thumbnail names are generated from the image names by appending the default suffix .thumb0.gif.

The -o option can be used to set the output directory, and the -s option can be used to change the suffix (for example, `.thumb0.jpg` for JPEG images). The -t option instructs the program to generate thumbnails using 24-bit color (true color). If the

output image format (for example, GIF) requires a colormap, a dynamic (not fixed) colormap is generated based on the input image colors.

Dumping Databases Using QbDumpDb

QbDumpDb dumps the contents of any Berkeley dbm file generated by the QBIC API. It dumps non-ASCII data in hexadecimal. The command syntax is:

```
QbDumpDb [-d databaseName] dbm_filename [-f featureName]
```

or

```
QbDumpDb -c catalogName -d databaseName -f feature1 [-f feature2 ...]
```

In the first syntax example, the dbm_filename should not contain any path information. If the dbm file is not in the current directory, you should provide the path information using the -d option. In the second syntax example, if you specify the -c option to indicate the catalog name and the -f option to indicate the feature name, QBIC will know what the feature table name is and will open and dump its contents for you.

If the Berkeley dbm file is a feature table, and the feature class name is also specified in the command line with an -f option, QbDumpDb can print the feature in a more readable format.

The -f option parameter should match the contents of the database, or the results will be unpredictable.

Extending tcl script with QBIC Functionality Using qbictcl

QBIC provides scripting capability by being integrated with the tcl scripting language. “qbictcl” includes the functionality of tcl’s standard function calls as well as QBIC function calls. These calls are described in “QBIC-Specific tcl Function Calls” on page 169.

“qbictcl” includes the functionality provided in QbMkDbs, QbQBE, QbMkThmb, and QdDumpDb with exactly the same function name and function syntax. Therefore, you can invoke these functions exactly the same way inside a tcl shell.

For example, to create and query `ibm33` catalog, you can enter the following in a qbictcl shell, which can be started by typing `qbictcl` in a command shell on the UNIX or NT platforms, or by clicking the `qbictcl` program on the Macintosh platform:

```
QbMkDbs -c ibm33 -d qbic/QbicData -f QbColorHistogramFeatureClass -l
image.dat

QbQBE -c ibm33 -d qbic/QbicData -f QbColorHistogramFeatureClass -i
flower01.jpg
```

You can also use qbictcl's lower-level functions for better control during image feature extraction. For example, the following script provides functionality similar to a standard database creation:

```
set qb [qbic_start ibm33 /qbic/QbicData w]  ;# Create a qbic descriptor
qbic_add_feature $qb QbColorHistogramFeatureClass ;# Add Qbic Feature
    to the descriptor
qbic_add_feature $qb QbDrawFeatureClass  ;#Add another Qbic Feature
qbic_add_image $qb flower01.jpg  ;# Extract features for flower01.jpg
qbic_add_image $qb flower02.jpg  ;# Extract features for flower02.jpg
...
qbic_end $qb  ;# delete the qbic descriptor
```

Setting QBIC Environment Variables

QBIC uses a set of optional environment variables to define search paths, default values, and so on for the engine. The following table lists the environment variables used by each program. Detailed descriptions of each environment variable follow the table.

NOTE: *The provided QBIC programs support the -e EnvVar=EnvValue option for setting the internal state of environment variables via the command line. The exception is QbDumpDb, where this is not needed.*

	Programs			
Environment Variables	QbMkDbs	QbQBE	QbMkThmb	qbictcl
QbicImagePath	X	X	X	X
QbicTextPath	X	X		X
QbicConnectMode	X	X		X
QbicDatabaseName	X	X		X
TCL_LIBRARY				X
QBTCL_LIB				X

The following are descriptions of QBIC environment variables:

Environment variable	Description
QbicImagePath	List of directories, separated by semi-colons, where QBIC will search for images to load. The default is the current directory.
QbicTextPath	List of directories, separated by semi-colons, where QBIC will search for a text file containing keyword information about a given image. The text file name must equal the image file name with the extension TXT. The default is the current directory.

Environment variable	Description
QbicDatabaseName	Initializes the name of the database to which a connection will be requested. If a name is not specified, the default name “QbicData” is used. The database name can be overridden using the -d command line option.
QbicConnectMode	Specifies the mode to connect to the database using the string “r” or “w” (for “read” and “write”). This environment variable is ignored if the call to QbGetConnection in QbCreate.cpp specifies the mode explicitly, as is done in all calls in the current implementation.
TCL_LIBRARY	Specifies the location where tcl or qbictcl can find init.tcl start script.
QBTCL_LIB	If you want to put the common script such as qbicmain-common-demo.tcl and qbicmain-common-index.tcl in a different directory from the cgi script ibm33.tcl (which may have to be moved to your cgi-bin directory). You need to set up this environment variable.

Modifying the tcl Script

The `qbic/script` directory contains a set of tcl scripts that qbictcl uses for database population and query. For example, `ibm33.tcl` is used both as a script for database population and as a cgi script for the web demo. You can edit the script to create and demonstrate other image collections. For example, suppose you have a collection of images located in `/myImage/foo` and the corresponding text (keyword) files are in `/myText/foo`, and you want to create a new catalog called bar. To do this, you would:

1. Copy `ibm33.tcl` to `bar.tcl`.
2. Use a text editor to make the following changes:
   - Change `set catalog ibm33` to `set catalog bar`
   - Change `set paths(image_root) $root/html/images/$catalog` to `set paths(image_root) /myImage/foo`
   - Change `set paths(text) $root/html/text/$catalog` to `set paths(text) /myText/foo`
   - If you want to put thumbnail images in a different directory, change the corresponding lines in `set paths(thumbnail)`.
3. To populate the database and generate thumbnails type:

```
qbictcl qbic-make bar.tcl
```

You can change the way the demo looks or behaves by changing the common script (`qbic/script/qbicmain-common-demo.tcl`) or by redefining variables and procedures in the cgi script such as `ibm33.tcl` or your `bar.tcl`.

For example, if you want the result table to display images in a 4-by-3 format, add the following code to the “user customizations” section of your template script:

```
set params(images_per_row) 4
set params(qbic_table_images) 12
```

If you want the “QBIC Information” page to look different, modify the page(info) section to suit your preferences. Any other changes are at your own risk.

Adding Your Feature Classes to the Demos

When you create your own feature classes, you must link them to QBIC’s executables.

For example, if you want to add MyFeatureClass to the QBIC demo, you would need to add the following code to your copy of the script, such as `ibm33.tcl`, after you build the catalog.

NOTE: *Descriptions of each variable follow the example.*

```
set type(MyFeatureClass) "My Feature Class"
set icon(MyFeatureClass) "letterM.gif"
set iconhelp(MyFeatureClass) "My Feature Class"
set fname(MyFeatureClass) "My+Feature+Class"
set feature(Average+Color) "MyFeatureClass"
set pickerpage(MyFeatureClass) "unenabled"
set pickertext(MyFeatureClass) "Unenabled"
```

The code in the example uses the following variables (as documented in `qbicmain-common-demo.tcl`):

- **type**–Maps the QBIC Feature class name to a name that appears in the interface.
- **icon**–Maps the QBIC Feature class name to a letter icon image. This is the icon that the user clicks to run the query. You may need to create your own icon image. The ones provided are 15-by-15 pixels with a black border.
- **iconhelp**– Maps the QBIC Feature class name to the text that appears at the top of the query panel.
- **fname**– Maps the QBIC Feature class name to a string posted in the URL query string.
- **feature**– Inverts the fname mapping by taking that part of the URL query string and mapping it back to a QBIC Feature class name.
- **pickerpage**–Set to “unenabled” so that nothing appears on the left part of the interface to enable a “picker” URL. You can modify this setting, but you will also have to make other major modifications. If you set this to “mypickerpage”, then you need to add a **page(mypickerpage)** variable and configure it appropriately.
- **pickertext**–Text that appears on the left part of the interface for the pickerpage.

See `qbicmain-common-demo.tcl` for more examples. If you were planning to use MyFeaturesClass with other QBIC applications, you should move the code to the common script.

QbWrapClass 6

QbWrapClass is a single, high-level class whose methods provide access to QBIC database population and query. To facilitate the integration of QBIC functionality into your applications, the QbWrapClass specifically implements several methods that have the same functionality as the command-line programs QbMkDbs, QbQBE, QbMkThmb, and QbDumpDb.

After you have experimented with these programs and are familiar with the command-line syntax used to execute these programs, you can readily incorporate the QBIC functionality into your application by creating a QbWrapClass object in your application and invoking the appropriate method. For example, the corresponding method for QbMkDbs is QbWrapDBS..

NOTE: *Because there is no command shell in Macintosh, start the qbictcl program, and invoke the function within the qbictcl shell.*

The following methods are described in this chapter:

You can see coding examples in Appendix C, “Sample Code” on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbicWrapClass method

This is the constructor for the class. This method has overloaded functions.

Syntax

```
QbicWrapClass( void )

QbicWrapClass( char *catName, char *dirName, char *mode )
```

Parameters

Input

- For the first method, none
- **catName**–For the second method, a pointer to a string that holds the name of the catalog

- **dirName**–For the second method, a pointer to a string that holds the name of the database
- **mode**–For the second method, a pointer to a string that holds the connection mode, which can be either “r” (read) or “w” (write)

Output

None

~QbicWrapClass method

This is the destructor for the class.

Syntax

```
~QbicWrapClass( void )
```

Parameters

None

DeleteFeatures method

This method deletes a feature class from the object database population or query.

Syntax

```
int DeleteFeatures (  char *featureName )
```

Parameters

Input

- **featureName**–Pointer of char to a string that holds the feature name

Output

Returns an integer. Successful if 0; otherwise, error.

GetFeatureDataImage method

This method computes the feature data for the input image or mask name.

Syntax

```
int GetFeatureDataImage (char *imgName, char *maskName, QbStringClass
    *str )
```

Parameters

Input

- **imgName**–Pointer to a string that holds the name of the input image
- **maskName**–Pointer to a string that holds the name of the input image mask, which can be NULL.
- **str**–Pointer to a QbStringClass object that will hold the feature data for the image or mask

Output

Returns an integer. Successful if 0; otherwise, error.

GetFeatureDataKey method

This method retrieves the feature data from the database for the input image key name.

Syntax

```
int GetFeatureDataKey (char *imgKey, QbStringClass *str )
```

Parameters

Input

- **imgKey**–Pointer to a string that holds the name of image key
- **str**–Pointer to a QbStringClass object that will hold the feature data for the key

Output

Returns an integer. Successful if 0; otherwise, error.

GetFeatureDataPicker method

This method retrieves the feature data for the picker image or mask name.

Syntax

```
int GetFeatureDataPicker(char *imgName, char *maskName, QbStringClass
    *str )
```

Parameters

Input

- **imgName**–Pointer to a string that holds the name of the input image

- **maskName**–Pointer to a string that holds the name of the input image mask, which can be NULL
- **str**–Pointer to a QbStringClass object that will hold the feature data for the image or mask.

Output

Returns an integer. Successful if 0; otherwise, error.

GetFeatureDataString method

This method computes the feature data for the input string description of the image.

Syntax

```
int GetFeatureDataString(char *strDes, QbStringClass *str )
```

Parameters

Input

- **strDes**–Pointer to a string that holds the input image description
- **str**–Pointer to a QbStringClass object that will hold the feature data for the image or mask

Output

Returns an integer. Successful if 0; otherwise, error.

GetParameters method

This method gets the parameter settings for the feature classes. Each feature class has its own parameter setting, either by default or by command line setting. This method retrieves the current settings for each class and puts them into the input QbStringClass object.

Syntax

```
int GetParameters (  QbStringClass *str )
```

Parameters

Input

str–Pointer to a QbStringClass object that will hold the parameters for each feature class that has already been added to the object (using the InsertFeatureClass method)

Output

Returns an integer. Successful if 0; otherwise, error.

GetParamForFeature method

This method gets the parameter class for a given feature class. Each feature class has its own parameter setting, either by default or by command line setting. This method retrieves the current parameter class for the class.

Syntax

```
QbParameterClass *GetParamForFeature(  char *featureName )
```

Parameters

Input

- **featureName**–Pointer to a string that holds the feature name

Output

Returns a pointer to a QbParameterClass object that holds the parameter for the feature. If NULL, there is an error.

InsertFeatures method

This method adds a feature class to the object for either database population or query.

Syntax

```
int InsertFeatures (  char *featureName )
```

Parameters

Input

- **featureName**–Pointer of char to a string that holds the feature name with possible optional parameters using the same syntax as the -f command line option in QbMkDbs or QbQBE.

Output

Returns an integer. Successful if 0; otherwise, error.

ListAllFeatures method

This method lists all feature classes in the current QBIC release

Syntax

```
char *ListAllFeatures(  void )
```

Parameters

Input

None.

Output

Returns a pointer to a string which contains all feature class names for the release. If NULL there is an error.

ListCatFeatures method

This method lists all feature classes in the catalog.

Syntax

```
char *ListCatFeatures(  void )
```

Parameters

Input

None

Output

Returns a pointer to a string that contains all feature class names for the catalog. If NULL, there is an error.

ListCatRecords method

This method obtains the number of images populated for a particular QBIC feature.

Syntax

```
int ListCatRecord( char *featureName )
```

Parameters

Input

- **featureName**–Pointer to a string that holds the feature name, such as QbColorHistogramFeatureClass

Output

Number of keys in the feature table.

QbicWrapClassConnect method

This method connects to the database, and opens the catalog using the default parameters. An overloaded function connects to the database, and opens the catalog using inputs.

Syntax

```
int QbicWrapClassConnect(  void )

int QbicWrapClassConnect(  char *catName, char *dirName, char *mode )
```

Parameters

Input

- For the first method, none
- **catName**–For the second method, a pointer of char to a string that holds the catalog name
- **dirName**–For the second method, a pointer of char to a string that holds the database name
- **mode**–For the second method, a pointer of char to a string that holds the connection mode, which can be either “r” (read) or “w” (write)

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapDBS method

This method implements the functionality of the command line program QbMkDbs. You can run QbMkDbs without any command-line arguments to display the command line syntax.

Syntax

```
int QbicWrapDBS( int argc, char **argv, QbStringClass *str=NULL )
```

Parameters

Input

- **argc**–Integer to indicate the dimension of the string array of the second input argument
- **argv**–Pointer to indicate a string array that holds the command-line arguments to the method
- **str**–Pointer to a QbStringClass object to store the output of the method. The default is NULL, in which case the output will not be returned. In many situations, the output will printed to screen. See “SetScreenPrint method” on page 48 for information.

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapDeleteImage method

This method removes the feature data from the QBIC database with the named key. The InsertFeatures method must have been called for this method to work.

Syntax

```
int QbicWrapDeleteImage(  char *imgName)
```

Parameters

Input

- **imgName**–Pointer of char to a string that holds the image key name

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapDeleteSubImage method

This method removes the feature data from the QBIC database with the named parent and child image keys. The InsertFeatures method must have been called for this method to work.

Syntax

```
int QbicWrapInsertImage(  char *imgNames )
```

Parameters

Input

- **imgNames**–Pointer of char to a string that holds the image names for both parent image and masks. The syntax should be `imageParent ( maskName1 maskName2 ... )`

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapDumpDb method

This method implements the functionality of the command-line program QbDumpDb. You can run QbDumpDb without any command-line argument to display the command-line syntax.

Syntax

```
int QbicWrapDumpDb( int argc, char **argv, FILE *stream )
```

Parameters

Input

- **argc**–Integer to indicate the dimension of the string array of the second input argument
- **argv**–Pointer to indicate a string array that holds the command line arguments to the method
- **stream**–Pointer to a file descriptor. The default is stdout. QbStringClass is not used to hold the output because the QbicWrapDumpDb method is often used to dump the feature data inside the database, which can be a lot of data that could easily exceed the available memory.

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapGetKeyWord method

This method retrieves keyword information, which is space-separated keywords stored in the object.

Syntax

```
char * QbicWrapGetKeyWord( void )
```

Parameters

Input

None

Output

Pointer to a string that holds the keyword information

QbicWrapInsertImage method

This method populates the QBIC database with the named image. The InsertFeatures method must have been called for this method to work.

Syntax

```
int QbicWrapInsertImage(  char *imgName)
```

Parameters

Input

- **imgName**–Pointer of char to a string that holds the image name

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapInsertSubImage method

This method populates the QBIC database with the named parent and mask images. The InsertFeatures method must have been called for this method to work.

Syntax

```
int QbicWrapInsertImage(  char *imgNames )
```

Parameters

Input

- **imgNames**–Pointer of char to a string that holds the image names for both parent image and masks. The syntax should be `imageParent ( maskName1 maskName2 ... )`

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapQBE method

This method implements the functionality of the command-line program QbQBE. You can run QbQBE without any command-line arguments to display the command line syntax.

Syntax

```
int QbicWrapQBE( int argc, char **argv, QbStringClass *str=NULL )
```

Parameters

Input

- **argc**–Integer to indicate the dimension of the string array of the second input argument
- **argv**–Pointer to indicate a string array that holds the command-line arguments to the method
- **str**–Pointer to a QbStringClass object to store the output of the method. The default is NULL, which means the output will not be returned. In many situations, the output will be printed to screen. See “SetScreenPrint method” on page 48 for more information.

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapQueryDB method

This method queries the data.

Syntax

```
int QbicWrapQueryDB(  QbStringClass *str )
```

Parameters

Input

- **str**–Pointer to a QbStringClass object that will hold the query result

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapQueryImage method

This method sets up a query specification using an input image and mask.

Syntax

```
int QbicWrapQueryImage(  char *imgName, char *maskName)
```

Parameters

Input

- **imgName**–Pointer to a string that holds the image name
- **maskName**–Pointer to a string that holds the mask name. This value can be NULL.

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapQueryKey method

This method sets up a query specification using an input key.

Syntax

```
int QbicWrapQueryKey(  char *keyName )
```

Parameters

Input

- **keyName**–Pointer to a string that holds the key name.

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapQueryPicker method

This method sets up a query specification using an input image key and image mask as a picker query.

Syntax

```
int QbicWrapQueryPicker(  char *keyName, char * maskName)
```

Parameters

Input

- **keyName**–Pointer to a string that holds the image key name
- **maskName**–Pointer to a string that holds the mask key name

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapQueryString method

This method sets up a query specification using an input image description string.

Syntax

```
int QbicWrapQueryString(  char *str )
```

Parameters

Input

- **str**–Pointer to a string that holds the image description.

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapRandomQueryDB method

This method returns random query keys.

Syntax

```
int QbicWrapRandomQueryDB(  QbStringClass *str, QbBoolean bool = True
    )
```

Parameters

Input

- **str**–Pointer to a QbStringClass object that will hold the query result
- **bool**–Boolean to indicate whether each time a call to this method will return a different set of random keys from the database. The default is True.

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapSetKeyWord method

This method sets the input keyword information to the object.

Syntax

```
void QbicWrapSetKeyWord( char *kw )
```

Parameters

Input

- **kw**–Pointer to a string that contains space-separated keywords

Output

None

QbicWrapSetPad method

When a thumbnail is generated, its default size is 100x100 pixels, which may not be the same aspect ratio as the input image. QBIC fills the extra regions with black pixels. Alternatively, you can generate a thumbnail with the same aspect ratio with 100 as the largest dimension. This method instructs the object on which method to use.

Syntax

```
void QbicWrapSetPad( QbBoolean pad )
```

Parameters

Input

- **bool**–Boolean value to indicate if padding is desired. Set to True to pad with black pixels.

Output

None

QbicWrapSetReturnedKeys method

This method sets the number of returned keys in a QBIC query. The default is 20.

Syntax

```
int QbicWrapSetReturnedKeys( int nhits )
```

Parameters

Input

- **nhits**–Integer to indicate how many keys you want QBIC to return in a query.

Output

An integer for the object's previous value.

QbicWrapSetThumb24Color method

This method instructs the object to generate the thumbnail in true color. The default is to generate the thumbnail using QBIC's color palette.

Syntax

```
void QbicWrapSetThumb24Color( QbBoolean bool )
```

Parameters

Input

- **bool**–Boolean value to indicate if true color is desired. Specify True for true color.

Output

None

QbicWrapThm method

This method implements the functionality of the command-line program QbMkThmb. You can run QbMkThmb without any command-line arguments to display the command line syntax.

Syntax

```
int QbicWrapThm( int argc, char **argv, QbStringClass *str=NULL )
```

Parameters

Input

- **argc**–Integer to indicate the dimension of the string array of the second input argument
- **argv**–Pointer to indicate a string array that holds the command-line arguments to the method
- **str**–Pointer to a QbStringClass object to store the output of the method. The default is NULL, in which case the output will not be returned. In many situations, the output will be printed to screen. See “SetScreenPrint method” on page 48 for information.

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapThumb method

This method generates thumbnails.

Syntax

```
int QbicWrapThumb( char *inImage, char *outImage)
```

Parameters

Input

- **inImage**–Pointer to a string that holds the input image name
- **outImage**–Pointer to a string that holds the output thumbnail name

Output

Returns an integer. Successful if 0; otherwise, error.

QbicWrapThumbXY method

This method sets the dimensions of the thumbnail. The default is 100x100 pixels.

Syntax

```
void QbicWrapThumbXY( int width, int height )
```

Parameters

Input

- **width**–Integer to indicate the new width of the thumbnail image, in pixels
- **height**–Integer to indicate the new height of the thumbnail image, in pixels

Output

None

SetScreenPrint method

This method instructs the object on whether to print output generated in methods, such as QbMkDbs, to the screen or not. The default is “1” (Yes).

Syntax

```
void SetScreenPrint( int sp )
```

Parameters

Input

- **sp**–Integer to indicate if output should be printed to screen. Use “1” for Yes, “0” for No.

Output

None

QbBaseClass 7

QbBaseClass is an abstract class at the top of the QBIC inheritance hierarchy. Derived classes can use this superclass' methods for defining the interface type and monitoring object status. This superclass only defines virtual functions so there is minimal overhead.

This class allows safe dynamic binding of C++ objects.

This class contains the public enumerated variable QbInterfaceType. Each of the interfaces possesses a unique interface type specified by QbInterfaceType:

- undefined
- datum
- genericData
- feature
- connect
- database
- keyDatabase
- databaseIterator
- parameter
- catalog
- query

The following methods are described in this chapter:

You can see coding examples in Appendix C, "Sample Code" on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbBaseClass method

This is the constructor for the class.

Syntax

```
QbBaseClass( void )
```

Parameters

None

~QbBaseClass method

This is the destructor for the class.

Syntax

```
virtual ~QbBaseClass( void )
```

Parameters

None

Checktype method

This method checks if the class name of the current object is the same as the given string className.

Syntax

```
virtual Checktype( const char *className )
```

Parameters

Input

className–A pointer to a string that holds the class name

Output

Boolean value; True if the current object has the same class name as className; False if the object has a different class name

IsOk method

This method returns the current state of the object. If an object encounters an internal error (such as a memory allocation failure in the constructor), the object's private variable, IamOK, is set to False. This method returns the current value of the variable IamOK.

Syntax

```
Boolean IsOk( void )
```

Parameters

Input

None

Output

A Boolean value: False if the object encountered internal errors; True otherwise

Itype method

This method returns the interface type.

Syntax

```
virtual const QbInterfaceType Itype( void )
```

Parameters

Input

None

Output

One of the QbInterfaceType enumerated types on page 49

Type method

This method returns the name of the derived class.

Syntax

```
virtual const char * Type( void )
```

Parameters

Input

None

Output

The name of the class, as a pointer to a constant string; for example QbColorHistogramFeatureClass.

QbCatalogClass

8

QbCatalogClass is derived from the QbBaseClass and handles all catalog-related operations, such as creating and deleting feature tables; adding, removing, and updating the feature data for input images in feature tables; and so on.

This release of the QBIC API uses Berkeley dbm files as persistent storage. Therefore, feature tables correspond to a set of files in the database directory. The catalog name is used as the file extension. For each new catalog, QbCatalogClass creates a feature index file, called FeatIdx, that keeps track of which QBIC features are indexed for the catalog and other global information for each feature class.

For each feature class, QbCatalogClass creates a feature table that uses part of the feature class name as the file name. Whenever possible, QBIC catalogs use the “8.3” file naming convention. The feature table name is created by first removing the leading “Qb” characters of the feature class name, and then joining the next 7 characters of the feature class name with an uppercase F. For example, for the QbColorHistogramFeatureClass, the table name will be ColorHiF. If the catalog name is ibm, then the full feature index name and the table name would be FeatIdx.ibm and ColorHiF.ibm respectively.

This class contains the public enumerated variable UpdateType which specifies whether to insert, increment, update, or force update records in the catalog:

- insertRecord
- incrementRecord
- updateRecord
- forceUpdateRecord

Restrictions:

- If insertRecord for an existing record fails, use updateRecord.
- If updateRecord for a record that does not exist in the table fails, use the forceUpdateRecord.
- insertRecord fails if any of the feature tables cannot insert the record. incrementRecord fails if none of the feature tables can insert the record.

Sometimes feature tables in the same catalog can become inconsistent (for example, one feature table might have more keys than another). This inconsistency can occur if the program was stopped in the middle of feature computation, or if a feature computation failed and exited prematurely. In this case, use the incrementRecord enumerated type to process the last few images to bring the tables to a consistent state.

The following methods are described in this chapter:

You can see coding examples in Appendix C, "Sample Code" on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbCatalogClass method

This is the constructor for the class.

Syntax

```
QbCatalogClass( const QbConnectClass &connect )
```

Parameters

Input

connect–A reference to the QbConnectClass instance

Output

None

~QbCatalogClass method

This is the destructor for the class.

Syntax

```
virtual ~QbCatalogClass( void )
```

Parameters

None

AddFeatureClass method

This method adds the feature class of the name given to the catalog. It creates a feature table for the class, and updates the feature index table (created by the CreateCatalog method) to include the added feature class.

Syntax

```
virtual int AddFeatureClass( const char *featureClassName )
```

Parameters

Input

featureClassName–A pointer to a string that holds the feature class name

Output

An integer value: 0 indicates success; nonzero indicates failure

AddRecord method

This method adds an object to the catalog. This method has overloaded functions:

- The first computes all current features classes and saves the information in persistent storage.
- The second assumes that the needed feature classes have already been created. It works when the client and server are in the same process space so that there is no need to pack any object information into a byte string and pass it to another process space to regenerate the object.

Syntax

```
virtual int AddRecord( const char *keyArg, const QbGenericDataClass
    &data, const QbParameterClass *parms, const UpdateType update =
    insertRecord, const char *featureListBuffer = NULL )

virtual int AddRecord (const char *keyArg, const QbGenericDataClass
    &data, const QbFeatureClass **features, const QbParameterClass
    **parms, int numOfFeatures, const UpdateType update = insertRecord)
```

Parameters

Input

- **keyArg**–A pointer to the key name for the object
- **data**–A reference to a QbGenericDataClass object to hold the raw data that describes the object for feature computation

- **parms**–For the first function, the parameters used to compute features; if NULL, use the feature's default parameter. For the second function, a pointer array of parameter classes; each parameter must exist (that is, they cannot be NULL)
- **features**–A pointer array of feature classes; each feature class must exist (that is, they cannot be NULL)
- **numOfFeatures**–An integer that contains the number of feature classes and parameters are in the previous two arguments (features and parms)
- **UpdateType**–One of the UpdateType enumerated variables on page 53 that specifies how data should be added to the record; the default is insertRecord
- **featureListBuffer**–A pointer to the feature names to be added to the feature data tables; if NULL, add to all tables

Output

An integer value: 0 indicates success; -1 indicates a fatal error; 1 indicates a non-fatal error

AddSample method

This method adds the image object into the given sampler table. Feature data is computed and saved in persistent storage, one for each sampler name in samplerList.

Syntax

```
virtual int AddSample( const char *keyArg, const char *samplerList,
   const QbGenericDataClass &data, const QbParameterClass *parms,
   const UpdateType update = insertRecord )
```

Parameters

Input

- **keyArg**–A pointer to the key for the sample
- **samplerList**–A pointer to the sampler list
- **data**–A reference to a QbGenericDataClass object that holds the sample to be added
- **parms**–A pointer to the parameters used to compute feature data; if NULL, use feature's default parameter
- **UpdateType**–One of the UpdateType enumerated variables on page 53 that sets the update type to insert

Output

An integer value: 0 indicates success; -1 indicates failure

CloseCatalog method

This method closes the catalog.

Syntax

```
virtual int CloseCatalog( void )
```

Parameters

Input

None

Output

An integer value: 0 indicates success; nonzero indicates failure

CreateCatalog method

This method creates the catalog in persistent storage (for example, on disk). This method creates a container in the persistent storage for global catalog information (such as feature names, global feature information, and so on).

The default implementation of this method uses Berkeley dbm as the database manager, so the CreateCatalog method creates the feature index table in the database directory with the name FeatIdx and with the catalog name as the extension. The feature index table stores global information such as feature names used for image feature extraction, number of images processed, global statistics of feature data, and so on.

Syntax

```
virtual int CreateCatalog( const char * catalogNameArg )
```

Parameters

Input

catalogNameArg–A pointer to a string that holds the catalog name

Output

An integer value: 0 indicates success; -1 indicates failure

CreateSampler method

This method creates a new sampler table in the database. A sampler table is not specific to a particular catalog. It can be used to put samples of feature data (presumably uniformly distributed) in the feature data space to allow a user to choose a particular place in the feature space to start a QBIC search.

Syntax

```
vitrual int CreateSampler( const char *samplerName, const char
   *featureclassName)
```

Parameters

Input

- **samplerName**–A pointer to a string that holds the sampler table name
- **featureclassName**–A pointer to a string that holds the feature class name

Output

An integer value: 0 indicates success; -1 indicates failure

DefineSubPart method

This method defines a subpart of a query object. Defining a subpart allows feature information to be computed for only a certain portion of an image that is of interest.

This method updates the subpart index table. It adds the input subpartKey to the child list of the parentKey, and adds the subpartKey with its mask data as a key-data pair entry.

Syntax

```
virtual int DefineSubPart( const char *parentKey, const char
   *subpartKey, const QbGenericDataClass &subpart )
```

Parameters

Input

- **parentKey**–A pointer to a string that holds the parent key (the input image name)
- **subpartKey**–A pointer to a string that holds the subpart key (the input mask image, also used as the child key)
- **subpart**–A reference to a QbGenericDataClass object that contains the raw image data and mask data

Output

An integer value: 0 indicates success; nonzero indicates failure

DeleteFeatureClass method

This method deletes a feature class object from the catalog. This deletes the feature data table from the database, and updates the feature table (created by the CreateCatalog method) to remove the deleted feature class.

Syntax

```
virtual int DeleteFeatureClass( const char * featureClassName )
```

Parameters

Input

featureClassName–A pointer to a string that holds the feature class name

Output

An integer value: 0 indicates success; nonzero indicates failure

DeleteRecord method

This method deletes a record from the catalog using the given key. This method has overloaded functions. The method assumes that the key is a parent key and that, therefore, deleting a parent will delete all logical children of that parent record.

Syntax

```
virtual int DeleteRecord( const char *keyArg, const char
    *featureListBuffer = NULL )

virtual int DeleteRecord( const char *keyArg, const QbFeatureClass
    **features, int numOfFeatures)
```

Parameters

Input

- **keyArg**–A pointer to the key name for the object
- **features**–A pointer array of feature classes; each feature class must exist (that is, they cannot be NULL)
- **numOfFeatures**–An integer that contains the number of feature classes and parameters in the previous two arguments (keyArg and features)

- **featureListBuffer**–A pointer to the feature names to be deleted from the feature data tables; if NULL, delete from all tables

Output

An integer value: 0 indicates success; -1 indicates a fatal error; 1 indicates a non-fatal error

DeleteSample method

This method deletes the named key from the list of samplers.

Syntax

```
virtual int DeleteSample( const char *key, const char *samplerList )
```

Parameters

Input

- **key**–A pointer to a string that holds the key for the sample
- **samplerList**–A pointer a string that holds a list of sampler names (separated with spaces) to be deleted

Output

An integer value: 0 indicates success; -1 indicates failure

DeleteSubPartRecord method

This method deletes only the sub-part of a record, leaving the parent record unaffected.

Syntax

```
virtual int DeleteSubPartRecord( const char * subKeyArg  )
```

Parameters

Input

subKeyArg–A pointer to a string that holds the name of the subkey

Output

An integer value: 0 indicates success; -1 indicates a fatal error; 1 indicates a non-fatal error

DeltaSinceDistTableBuild method

This method returns the number of records that have changed since the distance table was built. The distance table stores precomputed key-distance pairs for each image key in the feature data table. It can greatly speed up the QBIC search if you only need to find similar images within the catalog.

Syntax

```
virtual unsigned long DeltaSinceDistTableBuild( const char
   *featureClassname )
```

Parameters

Input

featureClassname–A pointer to the feature class name

Output

An unsigned long integer value representing the number of changed records

DeltaSinceIndexBuild method

This method returns the number of records that have changed since the index table was last built.

Syntax

```
virtual unsigned long DeltaSinceIndexBuild( const char
   *featureClassname )
```

Parameters

Input

featureClassname–A pointer to the feature class name

Output

An unsigned long integer value representing the number of changed records

DropCatalog method

This method drops (deletes) the catalog from persistent storage. Dropping the catalog removes all feature data tables and feature index tables from the database.

Syntax

```
virtual int DropCatalog( const char * catalogNameArg )
```

Parameters

Input

catalogNameArg–A pointer to a string that holds the catalog name

Output

An integer value: 0 indicates success; nonzero indicates failure

DropSampler method

This method drops a sampler table from the database.

Syntax

```
virtual int DropSampler( const char *samplerName )
```

Parameters

Input

samplerName–A pointer to the sampler table name

Output

An integer value: 0 indicates success; -1 indicates failure

GetDatabaseClassName method

This method returns the name of the database class. In the default implementation, the name is QbDbmDatabaseClass.

Syntax

```
virtual const char * GetDatabaseClassName( void )
```

Parameters

Input

None

Output

A pointer to a string that holds the name of the database class name

GetFeature method

This method gets the feature data of the given key from the catalog and uses that data to initialize the input feature class.

Syntax

```
virtual int GetFeature( const char *keyArg, QbFeatureClass &feature )
```

Parameters

Input

- **keyArg**–A pointer to a string that holds the name of the key for the record
- **feature**–A reference to the feature class instance

Output

An integer value: 0 indicates success; -1 indicates failure

GetFeatureDistTableName method

This method uses part of the feature name and the catalog name to construct the name for the table that holds precomputed distance data for the given feature. The name is assigned to the input tableName, a reference to QbStringClass type.

Syntax

```
int GetFeatureDistTableName( const char *featureName,  QbStringClass
   &tableName )
```

Parameters

Input

- **featureName**–A pointer to a string that holds the feature name
- **tableName**–A reference to a string to hold the table name

Output

An integer value: 0 indicates success; -1 indicates failure

GetFeatureIndexTableName method

This method uses part of the feature name and the catalog name to construct the name for the table that holds feature index data. The name is assigned to tableName of QbStringClass.

Syntax

```
int GetFeatureIndexTableName( const char *featureName, QbStringClass
   &tableName )
```

Parameters

Input

- **featureName**–A pointer to a string that holds the feature name
- **tableName**–A reference to a string to hold the table name

Output

An integer value: 0 indicates success; -1 indicates failure

GetFeatureSamplerTableName method

This method uses part of the feature name and the catalog name to construct the name for the table that holds sampler feature data. The name is assigned to tableName of QbStringClass.

Syntax

```
int GetFeatureSamplerTableName( const char *featureName, QbStringClass
   &tableName )
```

Parameters

Input

- **featureName**–A pointer to a string that holds the feature name
- **tableName**–A reference to a string to hold the table name

Output

An integer value: 0 indicates success; -1 indicates failure

GetFeatureTableName method

This method uses part of the feature class name and the catalog name to construct the name for the table that holds feature data. The name is assigned to tableName of QbStringClass.

Syntax

```
int GetFeatureTableName( const char *featureName, QbStringClass
   &tableName )
```

Parameters

Input

- **featureName**–A pointer to a string that holds the feature name
- **tableName**–A reference to a string to hold the table name

Output

An integer value: 0 indicates success; -1 indicates failure

GetGlobalFeatureInfo method

This method retrieves the global feature information for the given feature class name.

Syntax

```
virtual int GetGlobalFeatureInfo( const char *featureName,
   QbDatumClass &datum )
```

Parameters

Input

- **featureName**–A pointer to the feature class name
- **datum**–A reference to a QbDatumClass object whose byte string holds the global information

Output

An integer value: 0 indicates success; nonzero indicates failure

GetParentKey method

This method retrieves the parent key for a given subpart key.

Syntax

```
virtual int GetParentKey( const char *subpartKey, QbDatumClass
   &parentKey )
```

Parameters

Input

- **subpartKey**–A pointer a string that holds the names of the subpart key
- **parentKey**–A reference to a QbDatumClass object to hold the parent key

Output

An integer value: 0 indicates success; -1 indicates failure

GetRecord method

This method retrieves the feature data and feature class for a given key from persistent storage.

Syntax

```
virtual int GetRecord( const char *keyArg, const char
   *featureClassname, QbDatumClass &fdata )
```

Parameters

Input

- **keyArg**–A pointer to a string that holds the name of the key for the record
- **featureClassname**–A pointer to the feature class name
- **fdata**–A reference to a QbDatumClass object to hold the feature data of the given key

Output

An integer value: 0 indicates success; -1 indicates failure

GetSampleRecord method

This method retrieves the feature data of the named key and named sampler, and store the data in a datum object.

Syntax

```
virtual int GetSampleRecord( const char *key, const char *samplerName,
   QbDatumClass &datum )
```

Parameters

Input

- **key**–A pointer to a string that holds the name of the key for the sample
- **samplerName**–A pointer to the sampler name
- **datum**–A reference to a QbDatumClass object to hold the feature data

Output

An integer value: 0 indicates success; -1 indicates failure

GetSamplerFeature method

This method retrieves the feature data for the named key and the named sampler, and initializes the given feature object with the feature data.

Syntax

```
virtual int GetSamplerFeature( const char *key, const char
   *samplerName, QbFeatureClass &feature )
```

Parameters

Input

- **key**–A pointer to a string that holds the name of the key for the sample
- **samplerName**–A pointer to a string that holds the sampler name
- **feature**–A reference to a QbFeatureClass object to be initialized by the feature data

Output

An integer value: 0 indicates success; -1 indicates failure

GetSubPartDefinition method

This method retrieves the subpart definition (the mask) for the subpartKey.

Syntax

```
virtual int GetSubPartDefinition( const char *subpartKey,
   QbGenericDataClass &subpartData )
```

Parameters

Input

- **subpartKey**–A pointer a string that holds the name of the subpart key (the input mask image)
- **subpartData**–A reference to a QbGenericDataClass object to hold mask data

Output

An integer value: 0 indicates success; -1 indicates failure

GetSubPartKeys method

This method retrieves all subpart keys for the given parent key.

Syntax

```
virtual int GetSubPartKeys( const char *parentKey, unsigned int
   &numberOfKeys, QbDatumClass &subpartKeys )
```

Parameters

Input

- **parentKey**–A pointer to a string that holds the parent key
- **numberOfKeys**–A reference to an unsigned integer to hold the number of subkeys
- **subpartKeys**–A reference to a QbDatumClass object to hold the subpart keys, which are separated with spaces and are stored in a QbDatumClass object.

Output

An integer value: 0 indicates success; -1 indicates failure

Itype method

This method returns the catalog interface type.

Syntax

```
virtual const QbInterfaceType Itype( void )
```

Parameters

Input

None

Output

CatalogInterfaceType of QbInterfaceType defined in the QbBaseClass. See page 49.

ListFeatureClasses method

This method retrieves all available feature class names for which QBIC can compute feature data.

Syntax

```
virtual int ListFeatureClasses( QbDatumClass &featureList )
```

Parameters

Input

featureList–A reference to a QbDatumClass object whose byte string holds the feature class names, which are separated with spaces

Output

An integer value: 0 indicates success; -1 indicates failure

ListSampler method

This method retrieves the current sample names and their associated feature class names.

Syntax

```
virtual int ListSampler( QbDatumClass &samplerList )
```

Parameters

Input

samplerList–A reference to a QbDatumClass object to hold the sampler list

Output

An integer value: 0 indicates success; -1 indicates failure

NumberOfIndexRecords method

This method returns the number of image keys in the index table of the given feature in the catalog. The index table is used to enable fast searches.

Syntax

```
virtual unsigned long NumberOfIndexRecords( const char
   *featureClassname )
```

Parameters

Input

featureClassname–A pointer to a string that holds the feature class name

Output

An unsigned long integer value representing the number of image keys

NumberOfRecords method

This method returns the number of image keys the named feature class has computed.

Syntax

```
virtual unsigned long NumberOfRecords( const char *featureClassname )
```

Parameters

Input

featureClassname–A pointer to the feature class name

Output

An unsigned long integer value representing the number of image keys

NumberOfSamples method

This method returns the number of samples in a given sampler set.

Syntax

```
virtual unsigned long NumberOfSamples( const char *samplerName )
```

Parameters

Input

samplerName–A pointer to a string that holds the sampler name

Output

An unsigned long integer that represents the number of samples in the set

OpenCatalog method

This method checks if the database object has been created, and assigns the catalogName with the input catalogNameArg.

Syntax

```
virtual int OpenCatalog( const char * catalogNameArg )
```

Parameters

Input

catalogNameArg–A pointer to a string that holds the catalog name

Output

An integer value: 0 indicates success; nonzero indicates failure

ParentKnown method

This method returns True if the parent key is in the catalog. Otherwise, it returns False.

Syntax

```
virtual Boolean ParentKnown( const char *pKeyArg )
```

Parameters

Input

pKeyArg–A pointer to a string that holds the parent key

Output

A Boolean value: True if the parent key is in the catalog; False otherwise

PutGlobalFeatureInfo method

This method updates the global information in the feature table with the information stored in a QbDatumClass object.

Syntax

```
virtual int PutGlobalFeatureInfo( const char *featureName,
   QbDatumClass &datum )
```

Parameters

Input

- **featureName**–A pointer to a string that holds the feature class name
- **datum**–A reference to a QbDatumClass object that holds the global feature information for the given feature class

Output

An integer value: 0 indicates success; nonzero indicates failure

Type method

This method returns the name of the class.

Syntax

```
virtual const char * Type( void )
```

Parameters

Input

None

Output

A pointer to the string QbCatalogClass

QbConnectClass 9

QbConnectClass is derived from the QbBaseClass and is used to obtain a connection to a database or a file system. The Connect method establishes a connection to a database, or mounts a filesystem (processing that is expensive and typically only done once). The Disconnect method disconnects from the database, or dismounts the file system.

The function QbGetConnection in the file QbCreate.cpp implements the simple one-connection-per-address-space connection needed by the current QBIC implementation, which uses the Berkeley dbm system. QbGetConnection takes two arguments:

- Name of the database
- Name of the connection mode, which can be either “r” (read) or “w” (write)

It returns a const pointer to an instance of the connection class. Because it is a const, you cannot Disconnect or Connect to a database, nor can you delete the pointer yourself. However, you can create instances of a database appropriate for this connection using the functions CreateDatabaseClass and CreateKeyDatabaseClass. Use the method QbDestroyConnection, also in QbCreate.cpp, when you are finished with the connection. The default implementation for Berkeley dbm files creates a directory using the database name.

This chapter describes the following methods:

You can see coding examples in Appendix C, “Sample Code” on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbConnectClass method

This is the constructor for the class.

Syntax

```
QbConnectClass( void )
```

Parameters

None

~QbConnectClass method

This is the destructor for the class.

Syntax

```
virtual ~QbConnectClass( void )
```

Parameters

None

Connect method

This method connects to the named database in either read or write mode.

Syntax

```
virtual int Connect( const char *dsNameArg, const char *modeArg )
```

Parameters

Input

- **dsNameArg**–A pointer to a string containing the database instance name
- **modeArg**–A pointer to a string containing either “r” or “w”, to specify either read or write mode

Output

An integer value: 0 indicates success; -1 indicates failure

CreateCatalogClass method

This method provides an abstract architected method to create a catalog class.

Syntax

```
virtual QbCatalogClass * CreateCatalogClass( void )
```

Parameters

Input

None

Output

A pointer to a QbCatalogClass object

CreateDatabaseClass method

This method provides an abstract architected method to create a database class to support database operations such as Open, Close, Insert, Delete, Update, and so on.

Syntax

```
virtual QbDatabaseClass * CreateDatabaseClass( void )
```

Parameters

Input

None

Output

A pointer to a QbDatabaseClass object

CreateKeyDatabaseClass method

This method provides an abstract architected method to create a key database class for storing query results and for specifying the restriction list in a query.

Syntax

```
virtual QbKeyDatabaseClass * CreateKeyDatabaseClass( void )
```

Parameters

Input

None

Output

A pointer to a QbKeyDatabaseClass object

Disconnect method

This method disconnects from the database.

Syntax

```
virtual int Disconnect( void )
```

Parameters

Input

None

Output

An integer value: 0 indicates success; -1 indicates failure

GetConnectMode method

This method returns the database connection mode: either “r” or “w” for read or write.

Syntax

```
virtual const char * GetConnectMode( void )
```

Parameters

Input

None

Output

A pointer to a char that contains either “r” or “w” for read or write

GetDSName method

This method returns the name of the database to which the current object is connected.

Syntax

```
virtual const char * GetDSName( void )
```

Parameters

Input

None

Output

A pointer to a string that contains the database name

IsConnected method

This method returns True if the database is connected already, and False if it is not.

Syntax

```
virtual Boolean IsConnected( void )
```

Parameters

Input

None

Output

A Boolean value; True if connected, False it not

QbDatabaseClass

10

QbDatabaseClass is derived from the QbBaseClass and defines the abstract interface to a database to store feature data. You need to make a specific implementation for the virtual abstract functions (Open, Close, Insert, Update, Retrieve, Delete, Commit and CreateIterator) in order to use your database manager (IBM's UDB, Oracle, Sybase, dbm, and so on). In the current QBIC implementation, the Berkeley dbm system is used to implement the virtual functions.

Open and Close opens and closes a name space for subsequent operations. If the name space does not exist, Open creates it. Open also takes an optional second argument that specifies the mode, either "w" for write access or "r" for read access. The default is to open using the same mode supported by the connection to the database. However, a call to Open with the "w" (write) mode when the connection uses the read mode will fail.

To remove the namespace use the CloseAndRemove member function. The Insert member function inserts the key/value pair into the database and fails if the key already exists. To update an existing record, use the Update function. Update fails if the key does not exist. Given a key, Retrieve returns the value and Delete deletes the record. Because the database object often keeps state, copying of QbDatabaseClass is not allowed. If multiple objects are needed, the program is free to create them. CreateIterator is an abstract virtual function that creates an instance of a database iterator.

This chapter describes the following methods:

- “Retrieve method” on page 87
- “Update method” on page 87

Create a QbDatabaseClass object by calling the CreateDatabaseClass method of the QbConnectClass.

You can see coding examples in Appendix C, “Sample Code” on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbDatabaseClass method

This is the constructor for the class. This method attaches to the database connection specified.

Syntax

```
QbDatabaseClass( const QbConnectClass &connectArg )
```

Parameters

Input

connectArg–A reference to a QbConnectClass object that specifies the database connection

Output

None

~QbDatabaseClass method

This is the destructor for the class.

Syntax

```
virtual ~QbDatabaseClass( void )
```

Parameters

None

Close method

This method closes an open database table or file.

Syntax

```
virtual int Close( void )
```

Parameters

Input

None

Output

An integer value: 0 indicates success; -1 indicates failure

CloseAndRemove method

This method closes an open database table or file, and drops the table or deletes the file.

Syntax

```
virtual int CloseAndRemove( void )
```

Parameters

Input

None

Output

An integer value: 0 indicates success; -1 indicates failure

Commit method

This method allows a database that has transaction support to keep a record of all the updates, additions, and deletions that have been made. In other words, this method tells the database to commit the changes to the database.

Syntax

```
virtual int Commit( void )
```

Parameters

Input

None

Output

An integer value: 0 indicates success; -1 indicates failure

CreateIterator method

This method creates an iterator object for the database.

Syntax

```
virtual QbDatabaseIteratorClass * CreateIterator( void )
```

Parameters

Input

None

Output

A pointer to a QbDatabaseIteratorClass object; return NULL if the database is not yet open

Delete method

This method deletes a record from the database.

Syntax

```
virtual int Delete( const QbDatumClass &key )
```

Parameters

Input

key–A reference to a QbDatumClass object that contains the key

Output

An integer value: 0 indicates success; -1 indicates failure

GetConnection method

This method returns a reference to the database connection specified by protected member connection.

Syntax

```
const QbConnectClass & GetConnection( void )
```

Parameters

Input

None

Output

A reference to a QbConnectClass object

GetContainerName method

This method returns the name of the database that is opened.

Syntax

```
virtual const char * GetContainerName( void )
```

Parameters

Input

None

Output

A pointer to a string to hold the container name

Insert method

This method inserts a record into the database.

Syntax

```
virtual int Insert( const QbDatumClass &key, const QbDatumClass &value
    )
```

Parameters

Input

key–A reference to a QbDatumClass object that contains the key

value–A reference to a QbDatumClass object that contains the record value

Output

An integer value: 0 indicates success; 1 indicates the record already exists; -1 indicates a fatal error

IsEmpty method

This method determines whether the database is empty.

Syntax

```
virtual int IsEmpty( void )
```

Parameters

Input

None

Output

An integer value: 0 indicates the database is empty; 1 indicates the database is not empty; -1 indicates a fatal error

Open method

This method opens a database table or file. It creates one if the named one does not exist. The argument mode specifies whether to open the database in read or write mode.

The Open method fails if the mode specified is write while the mode of the database connection is read.

Syntax

```
virtual int Open( const char *tableOrFileName, const char *mode )
```

Parameters

Input

- **tableOrFileName**–A pointer to a string that contains the database table or file name
- **mode**–A pointer to a string containing “r” or “w” to represent the mode (read or write)

Output

An integer value: 0 indicates success; -1 indicates failure

Retrieve method

This method reads a record from the database.

Syntax

```
virtual int Retrieve( const QbDatumClass &key, const QbDatumClass
   &value )
```

Parameters

Input

- **key**–A reference to a QbDatumClass object that contains the key
- **value**–A reference to a QbDatumClass object that contains the record value

Output

An integer value: 0 indicates success; 1 indicates a nonfatal error (such as trying to read a nonexistent record); -1 indicates a fatal error

Update method

This method updates a record in the database.

Set the insert flag to True to insert the record if it does not exist. Set the insert flag to False to return an error if the record does not exist (rather than inserting it.

Syntax

```
virtual int Update( const QbDatumClass &key, const QbDatumClass
   &value, const Boolean insert = False )
```

Parameters

Input

- **key**–A reference to a QbDatumClass object that contains the key
- **value**–A reference to a QbDatumClass object that contains the value
- **insert**–A Boolean value: True indicates that nonexistent records are to be inserted; False indicates that this method will not insert nonexistent records, but will return an error

Output

An integer value: 0 indicates success; 1 indicates a nonfatal error (such as trying to update a nonexistent record with the insert flag set to False); -1 indicates a fatal error

QbDatumClass

11

QbDatumClass is derived from the QbBaseClass and contains a pointer to a byte string, the length of the byte string, and a character description of the format. The Get and Set methods allow you to get and set the pointer and size variables. The GetFormatInfo and SetFormatInfo methods allow you to access format information.

This class supports marking of allocated memory so it can be deleted, freed, or left alone in the destructor, or reassigned using Set. It also has an assignment operator.

QbDatumClass derives from QbBaseClass and stores key-value pairs.

This method contains the following protected member variables:

- A byte string (unsigned char *bytes)
- Its size (unsigned long size)
- An enum (storageEnum) that describes the storage format of the byte string:
  - leaveAlone
  - mallocStorage
  - newStorage
  - newArrayStorage
  - QbNewStorage

In this class, the = operator has been overloaded to copy both data and format information from one QbDatumClass object to another. The syntax to copy a QbDatumClass object is as follows:

```
QbDatumClass &operator = (const QbDatumClass &in);
```

This chapter describes the following methods:

You can see coding examples in Appendix C, “Sample Code” on page 185. Other programs are located in the qbic/QbicApi directory.

QbDatumClass method

This is the constructor for the class. It sets size to zero and storageEnum type to leaveAlone.

Syntax

```
QbDatumClass( void )
```

Parameters

None

~QbDatumClass method

This is the destructor for the class.

Syntax

```
virtual ~QbDatumClass( void )
```

Parameters

None

Get method

This method retrieves the byte string and the size of an object. This method has three overloaded functions.

Syntax

```
virtual void Get( unsigned long &n, unsigned char *&p )
virtual void Get( unsigned long &n, char *&p )
virtual void Get( unsigned long &n, void *&p )
```

Parameters

Input

- **n**–A reference to an unsigned long integer that represents the length the byte string
- **p**–For the first function, a pointer to a reference to an unsigned char that will hold the byte string

- **p**–For the second function, a pointer to a reference to a char that will hold the byte string
- **p**–For the third function, a pointer to a reference to a void that will hold the byte string

Output

None

GetFormatInfo method

This method returns a string containing the format information that was set up by the SetFormatInfo method.

Syntax

```
virtual const char * GetFormatInfo( void )
```

Parameters

Input

None

Output

A pointer to a constant char that describes the format information.

Itype method

This method returns the interface type defined in the base class.

Syntax

```
virtual const QbInterfaceType Itype( void )
```

Parameters

Input

None

Output

DatumInterfaceType of QbInterfaceType defined in the QbBaseClass. See page 49.

Set method

This method cleans up any data already in the object, then reinitializes the object based on the input values. This method deletes storage that is pointed to by the variable pArg if its storageType enum is not leaveAlone.

Syntax

```
virtual void Set( const unsigned long nArg, unsigned char *pArg, const
    storageEnum storageType = leaveAlone )
```

Parameters

Input

- **nArg**–An unsigned long integer that represents the length the byte string of the second argument
- **pArg**–A pointer of unsigned char that describes the input data for the object
- **storageType**–An enumerated type, described on page 89, that describes how the second argument was created; the default is leaveAlone

Output

None

SetFormatInfo method

This method sets format information which can be used, for example, to describe the format of a byte string or feature type.

Syntax

```
virtual void SetFormatInfo( char *f, const storageEnum storageType =
    leaveAlone )
```

Parameters

Input

- **f**–A pointer to a string that holds format information for the byte string
- **storageType**–An enumerated type, described on page 89, that describes how the first argument was created; the default is leaveAlone.

Output

None

Type method

This method returns a constant string of the class name.

Syntax

```
virtual const char * Type( void )
```

Parameters

Input

None

Output

A constant string, QbDatumClass

QbDbIteratorClass 12

QbDbIteratorClass is derived from the QbBaseClass. It is designed to define the interface for a database iterator. It has three virtual methods: More, Next, and Reset (these methods are similar to the OQS iterator functions). The More method returns true if there is more data. The Next method returns the next record. The Reset method resets the iterator so that a future call to Next returns the first record instead of the next one.

This class contains the public enumerated variable nextEnum which indicates the desired cursor position:

- FirstKey
- NextKey
- ThisKey

To create an instance of QbDbIteratorClass, use the QbDatabaseClass method, CreateIterator. Using this method ensures that the iterator is consistent with the database. The state of an iterator is undefined if an Insert, Delete, or Update is done to the database after the iterator is created.

When you no longer need the iterator, delete it to avoid memory leaks.

This chapter describes the following methods:

You can see coding examples in Appendix C, “Sample Code” on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbDbIteratorClass method

This is the constructor for the class.

Syntax

```
QbDbIteratorClass( void )
```

Parameters

None

~QbDbIteratorClass method

This is the destructor for the class.

Syntax

```
virtual ~QbDbIteratorClass( void )
```

Parameters

None

More method

This method returns 1 if there is more data, and 0 if this is the last item in the database.

Syntax

```
virtual int More( void )
```

Parameters

Input

None

Output

An integer value: 0 indicates the last item in the database; 1 indicates that there is more data

Next method

This method advances to the next record in the database (or to the first record following the Reset method). This method copies key and value pairs of the record to key and value objects of QbDatumClass.

Syntax

```
virtual int Next( QbDatumClass &key, QbDatumClass &value, nextEnum
    nextFlag = NextKey )
```

Parameters

Input

- **key**–A reference to a QbDatumClass object that holds the current key
- **value**–A reference to a QbDatumClass object that holds the current value associated with the key
- **nextFlag**–One of the nextEnum enumerated variables on page 95 that specifies the position of the cursor. The default is next record.

Output

An integer value: 0 indicates success; -1 indicates failure

Reset method

This method resets the iterator so that the first call to the Next method returns the first value in the database.

Syntax

```
virtual int Reset( void )
```

Parameters

Input

None

Output

An integer value: 0 indicates success; -1 indicates failure

QbFeatureClass

13

QbFeatureClass is derived from the QbBaseClass and is used to represent a feature. The methods ComputeFeatures, Distance, ExecutionCostForDistanceFunction, FilterResult, and ExecutationCostForFilterFunction are the required virtual functions.

The ComputeFeature method takes two arguments, a pointer to a QbGenericDataClass object and a pointer to a QbParameterClass object that contains the query specification.

The Distance method returns the distance between the current instance of a feature class and another instance of the feature class.

The FilterResult method arguments are a connected database, an optional key sublist of this database, the number of results requested by the query, the name of the catalog, and the query parameters. This method can return a ranked or an unranked list. If the list is not ranked, the returned distances do not reflect final query ordering, and the calling method or function should use the Distance method to determine the final ordering.

The GenericDataClassname and ParameterClassname methods return the name of the QbGenericDataClass and QbParameterClass instances needed by this feature.

The GetDefaultParameters method returns a pointer to the default parameters for this feature class. This default parameter data is normally implemented as class static member data. The return type is const so that the caller cannot delete the result. To change the default parameters, make a copy of the default parameters using the assignment operator before changing the values.

The UpdateGlobalFeatureInfo method allows the implementor of a feature to keep various statistics, such as sum and sum of squares that are needed for distance computations. These statistics allow implementation of such algorithms as Mahalanobis distance. The base implementation of this method implements a simple counter to count the number of records in a database. You can use the parameter information to determine when to use global information, using the ParameterUpdateFromGlobalInfo method.

Currently, only the Texture feature uses a variance-weighted distance. In this case, the sum and sum of squares are stored as the global information for the feature. During a query, a parameter value of `Vxx` (where `V` stands for variance and `xx` is an optional numerical weight) implements an inverse variance weighting. That is, a given feature is weighted by a coefficient given as one divided by the variance of that feature value for all data in the collection times the weight `xx`. For example, V5 means (5/variance)

weighting. During a query, the global information is read from the catalog and used to update the weights that will be used in the distance computation.

NOTE: *QBIC has no knowledge of the type of weighting or how the distance will be computed. This is left entirely up to the feature designer.*

Features must be stored in a database. This is accomplished using the methods ToByteString and FromByteString. These methods convert the contents of the class into a byte string and set the contents of the class from a byte string. The argument for both of these methods is a reference to a QbDatumClass instance.

A utility method called FeatureSize returns an unsigned long that describes the length of the feature vector needed by ToByteString.

This class contains the public enumerated variable GlobalFeatureInfoEnum:

- insert
- delete
- indexInsert
- indexDelete
- indexReset
- indexIncrReset
- deltaSinceIndex
- distTableReset
- deltaSinceDistTable

This chapter describes the following methods:

You can see coding examples in Appendix C, "Sample Code" on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbFeatureClass method

This is the constructor for the class.

Syntax

```
QbFeatureClass( void )
```

Parameters

None

~QbFeatureClass method

This is the destructor for the class.

Syntax

```
virtual ~QbFeatureClass( void )
```

Parameters

None

ComputeFeatures method

This method converts generic data to features. Given generic data and parameters, this method computes feature data based on the input that is pointed to by the QbGenericDataClass object. The QbGenericDataClass object can contain either raw data or a key to retrieve the data from a database.

Syntax

```
virtual int ComputeFeatures( const QbGenericDataClass *data, const
   QbParameterClass *parms )
```

Parameters

Input

- **data**–A pointer to a QbGenericDataClass object
- **parms**–A pointer to a QbParameterClass object that contains the parameters for the computation

Output

An integer value: 0 indicates success; nonzero indicates failure

Distance method

This method computes the distance between this feature instance (typically the query feature) and another instance of the same feature using the parameters passed in the second argument, parms. The abortDistance argument can be used to speed up distance computation. If, in the middle of the distance computation, you are certain that the final distance exceeds the abortDistance, you can stop the computation and return it with the abortDistance value.

Syntax

```
virtual fastfloat Distance( const QbFeatureClass *feature, const
    QbParameterClass *parms, const float  abortDistance )
```

Parameters

Input

- **feature**–A pointer to the QbFeatureClass object
- **parms**–A pointer to a QbParameterClass object that contains the parameters for the computation
- **abortDistance**–A float number for the abort distance

Output

A fastfloat (double) value between 0 and 1; -1 indicates an error

ExecutionCostForDistanceFunction method

This method provides a measure of the complexity of the distance evaluation. Such a measure can be used to choose the optimized parameter for running a query. The default implementation simply returns -1.0.

Syntax

```
virtual float ExecutionCostForDistanceFunction( const QbFeatureClass
   *feature, const QbParameterClass *parms, const float abortDistance
   )
```

Parameters

Input

- **feature**–A pointer to the feature instance
- **parms**–A pointer to an object that contains the parameters for the computation
- **abortDistance**–A float for the abort distance

Output

A float value between 0 and 1; -1 indicates an error

ExecutionCostForFilterFunction method

This method provides a measure of the complexity of the FilterResult method. The default implementation simply returns -1.0.

Syntax

```
virtual float ExecutionCostForFilterFunction( QbDatabaseClass
   &database, const QbKeyDatabaseClass *list, const unsigned long
   queryResultsRequested, const char *catalogName, const
   QbParameterClass *parms )
```

Parameters

Input

- **database**–A reference to a QbDatabaseClass object
- **list**–A pointer to a QbKeyDatabaseClass object that holds a list of key and distance pairs
- **queryResultsRequested**–An unsigned long integer that represents how many items are needed
- **catalogName**–A pointer to a string that holds the catalog name
- **parms**–A pointer to a QbParameterClass object that contains the parameters for the computation

Output

A float value between 0 and 1; -1 indicates an error

FeatureSize method

This method returns the number of bytes of the feature vector.

Syntax

```
virtual unsigned long FeatureSize( void )
```

Parameters

Input

None

Output

An unsigned long integer value that represents the number of bytes in the feature vector

FilterResult method

This method produces a restriction list, which can be either ranked or unranked. The default result is -1, which indicates that no restriction list is produced.

Syntax

```
virtual int FilterResult( QbDatabaseClass &database, const
   QbKeyDatabaseClass *list, const unsigned long
   queryResultsRequested, const char *catalogName, const
   QbParameterClass *parms, QbKeyDatabaseClass &results )
```

Parameters

Input

- **database**–A reference to a QbDatabaseClass object; this object cannot be NULL
- **list**–A pointer to a QbKeyDatabaseClass object that contains a list of key and distance pairs; if this list is NULL, it means there are no restrictions
- **queryResultsRequested**–An unsigned long integer that represents how many items are needed
- **catalogName**–A pointer to the catalog name
- **parms**–A pointer to a QbParameterClass object that contains the parameters for the computation
- **result**–A reference to a QbKeyDatabaseClass object to hold the result

Output

An integer value; 0 indicates that a list was produced; -1 indicates no list was produced

FromByteString method

This method decodes a byte string from the given data, and initializes an object with that decoded data.

Syntax

```
virtual int FromByteString( const QbDatumClass &data )
```

Parameters

Input

data–A reference to a QbDatumClass object that holds the data to initialize the feature class object

Output

An integer value: 0 indicates success; -1 indicates failure

GenericDataClassname method

This method returns the QbGenericDataClassname, the QbGenericImageDataClassname, or whatever the current feature class expects in order to compute feature data.

Syntax

```
virtual const char * GenericDataClassname( void )
```

Parameters

Input

None

Output

A pointer to a string that contains the QbGenericDataClass or its derived class name

GetDefaultParameters method

This method returns the default parameters for the feature class.

Syntax

```
virtual const QbParameterClass * GetDefaultParameters( void )
```

Parameters

Input

None

Output

A pointer to a QbParameterClass object that contains the default parameters

GetDimension method

This method returns the dimension of the feature vector.

Syntax

```
virtual const unsigned long GetDimension( void )
```

Parameters

Input

None

Output

An unsigned long integer that represent the dimension of the feature vector

IsSubPartFeature method

This method returns whether the object is a subpart feature.

Syntax

```
virtual Boolean IsSubPartFeature( void )
```

Parameters

Input

None

Output

A Boolean value. True if it is a subpart; False if it is not

ParameterClassname method

This method returns the name of the QbParameterClass object that this feature supports.

Syntax

```
virtual const char * ParameterClassname( void )
```

Parameters

Input

None

Output

A pointer to a string that contains the parameter class name

ParameterUpdateFromGlobalInfo method

This method updates the feature's parameters using the global information for the feature. In the QbTextureFeatureClass, for example, the distance calculation requires the global statistics of the variance of the texture feature vector in the given catalog in the database. These global statistics can be stored in the global information for the catalog and can be retrieved. When you have the global information, the ParameterUpdateGlobalInfo method can update the parameter class, which can be subsequently used as the input parameter for feature distance computations.

Syntax

```
virtual int ParameterUpdateFromGlobalInfo( QbParameterClass &parms,
   const QbDatumClass &globalInfo )
```

Parameters

Input

- **parms**–A reference to a QbParameterClass object that contains the parameters for the computation
- **globalInfo**–A reference to a QbDatumClass object that contains the global information for the feature

Output

An integer value: 0 indicates success; -1 indicates failure

ToAsciiString method

This method converts the feature data into a formatted ASCII string and assigns it to the byte string of the input string, a reference to a QbDatumClass object.

Syntax

```
virtual int ToAsciiString( C &string )
```

Parameters

Input

string–A reference to a QbDatumClass object that stores the feature data

Output

An integer value: 0 indicates success; -1 indicates failure

ToByteString method

This method encodes feature information to a byte string which is assigned to data.

Syntax

```
virtual int ToByteString( QbDatumClass &data )
```

Parameters

Input

data–A reference to a QbDatumClass object to hold the feature data

Output

An integer value: 0 indicates success; -1 indicates failure

UpdateGlobalFeatureInfo method

This method updates the global information in a QbDatumClass object based on the type (a public enumerated type defined within the class).

Syntax

```
virtual int UpdateGlobalFeatureInfo( QbDatumClass &datum, const
   GlobalFeatureInfoEnum gType)
```

Parameters

Input

- **datum**–A reference to a QbDatumClass object that holds the global information to be updated
- **gType**–An enumerated type of GlobalFeatureInfoEnum, described on page 100.

Output

An integer value: 0 indicates success; -1 indicates failure

QbGenericDataClass 14

QbGenericDataClass is derived from the QbBaseClass and defines generic data: images, video, text, and audio. This class encapsulates the data between the feature extraction process, the user interface, and external data (image files). This class is fully implemented and gives you a standardized way to pass “query by example” or “query from sampler data” between a client and server. It also provides a standard way of passing domain-specific information between a client and server. This class does nothing to document the format of the domain-specific data. To do this, you need to subclass this class and specify the domain-specific data.

An example of a subclass of this class is the QbGenericImageDataClass class, which contains domain-specific image data.

You pass it to the ComputeFeature method in any class that derives from QbFeatureClass (such as QbColorHistogramFeatureClass), as input for feature computation. A protected member variable, dataSource type, in QbGenericDataClass determines how the ComputeFeature method should proceed in feature extraction.

dataSource is a public enumerated type that takes the following values:

- **unknown**–Indicates that the data is not available
- **memory**–Indicates that the data is already in memory, so the feature data can be readily computed
- **sample**–Indicates that the needed feature data is stored in a special sampler database and should be retrieved from there
- **example**–Indicates that the needed feature data is stored in the database and should be retrieved from there
- **picker**–Indicates that the data is a special “picker” type already in memory, so the feature data can be readily computed

Both QbGenericDataClass and QbParameterClass implement ToQueryString and FromQueryString methods. These methods are used to convert the class to an ASCII query string to allow the query to be passed from the client to a server query engine that can be physically separated, even on a different platform.

This chapter describes the following methods:

You can see coding examples in Appendix C, “Sample Code” on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbGenericDataClass method

This is the constructor for the class.

Syntax

```
QbGenericDataClass( void )
```

Parameters

None

~QbGenericDataClass method

This is the destructor for the class.

Syntax

```
virtual ~QbGenericDataClass( void )
```

Parameters

None

DataIsSubPartDefinition method

This method sets a flag that indicates that the object contains subpart data.

Syntax

```
void DataIsSubPartDefinition( void )
```

Parameters

None

DecodeSubPartDef method

This method decodes a subpart definition from input to initialize the class object.

Syntax

```
virtual int DecodeSubPartDef( const QbDatumClass &subPartDatum )
```

Parameters

Input

subPartDatum–A reference to a QbDatumClass object that holds the subpart description to be decoded

Output

An integer value: 0 indicates success; -1 indicates failure

EncodeSubPartDef method

This method encodes the current object's subpart description into the byte string of the input subPartDatum.

Syntax

```
virtual int EncodeSubPartDef( QbDatumClass &subPartDatum )
```

Parameters

Input

subPartDatum–A reference to a QbDatumClass object that will hold the byte string of the encoded subpart description.

Output

An integer value: 0 indicates success; -1 indicates failure

FromQueryString method

This method initializes the current object with the date encoded in the byte string of the input QbDatumClass object, datum.

Syntax

```
virtual int FromQueryString( const QbDatumClass &datum, const char
    *featureClassname )
```

Parameters

Input

- **datum**–A reference to a QbDatumClass object that holds the encoded generic data
- **featureClassname**–A pointer to a string that contains the feature class name, such as QbColorHistogramFeatureClass

Output

An integer value: 0 indicates success; -1 indicates failure

GetCatalog method

This method returns the catalog name.

Syntax

```
const char * GetCatalog( void )
```

Parameters

Input

None

Output

A pointer to a string to hold the catalog name

GetData method

This method returns a reference to a QbDatumClass object packed with feature data.

Syntax

```
const QbDatumClass & GetData( void )
```

Parameters

Input

None

Output

A reference to a QbDatumClass object that packs the feature data in its byte string.

GetDerivedClassData method

This method returns a reference to a QbDatumClass object that packs domain-specific data, such as image, text, and audio, into its member byte string.

Syntax

```
const QbDatumClass & GetDerivedClassData( void )
```

Parameters

Input

None

Output

A reference to a QbDatumClass object that packs the domain-specific data into its member byte string for feature computation

GetIsQuery method

This method returns True if the object corresponds to query data description. In some feature classes, like QbColorHistogramFeatureClass, the ComputeFeature method might need additional processing on query feature data to increase the speed of the query.

Syntax

```
Boolean GetIsQuery( void )
```

Parameters

Input

None

Output

A Boolean value; True if the object corresponds to query data description

GetKey method

This method returns the name of the database key.

Syntax

```
const char * GetKey( void )
```

Parameters

Input

None

Output

A pointer to a string that holds the name of the database key

GetSampler method

This method returns the name of the sampler.

Syntax

```
const char * GetSampler( void )
```

Parameters

Input

None

Output

A pointer to a string to hold the sampler name

GetTextDescription method

This method returns the text description of the generic data.

Syntax

```
char * GetTextDescription( void )
```

Parameters

Input

None

Output

A pointer to a string to hold the text description

GetType method

This method returns the data source type of an object.

Syntax

```
const dataSource GetType( void )
```

Parameters

Input

None

Output

An enumerated type of dataSource, described on page 111

isSubPartDefinition method

This method returns a value that represents whether the generic data is in a subpart definition. For an image, for example, you can pass a mask image file with the same size as the original image. The mask file has pixel values of either zero or non-zero. QBIC performs feature computations on pixels of the image where the mask has non-zero pixel values.

Syntax

```
int isSubPartDefinition( void )
```

Parameters

Input

None

Output

An integer value: 1 if the generic data is in a subpart description, 0 if not

Itype method

This method returns the interface type defined in the base class.

Syntax

```
virtual const QbInterfaceType Itype( void )
```

Parameters

Input

None

Output

GenericDataInterfaceType of QbInterfaceType defined in the QbBaseClass. See page 49.

MakeQueryByExampleObject method

This method defines the object as a query-by-example type for the given key and catalog names. Query by example means that the feature data is already in the database. Therefore, by passing the catalog name and the key, feature data can be extracted from the database.

Syntax

```
int MakeQueryByExampleObject( const char *CatalogName, const char
   *KeyName )
```

Parameters

Input

- **CatalogName**–A pointer to a string that holds the catalog name
- **KeyName**–A pointer to a string that holds the key name

Output

An integer value: 0 indicates success; -1 indicates failure.

MakeQueryByTextObject method

This method defines the object as a query-by-text type. Query-by-text searches through keywords to find records with the same keyword.

Syntax

```
int MakeQueryByTextObject( const char *text )
```

Parameters

Input

text–A pointer to a string that holds the text or keyword description

Output

An integer value: 0 indicates success; -1 indicates failure

SetCatalog method

This method deletes the current catalog name, and replaces it with the new one pointed to by catalogArg.

Syntax

```
int SetCatalog( const char *catalogArg )
```

Parameters

Input

catalogArg–A pointer to a string that holds the new catalog name

Output

An integer value: 0 indicates success; -1 indicates failure

SetIsQuery method

This method sets the object as a query object if bArg=True. See also “GetIsQuery method” on page 116.

Syntax

```
Boolean SetIsQuery( Boolean bArg )
```

Parameters

Input

A Boolean value.

Output

A Boolean value that has the same value returned by a GetIsQuery call before the new value bArg is set.

SetKey method

This method deletes the current key name, and replaces it with the new one pointed to by keyArg.

Syntax

```
int setKey( const char *keyArg )
```

Parameters

Input

keyArg–A pointer to a string that holds the key name

Output

An integer value: 0 indicates success; -1 indicates failure

SetSampler method

This method deletes the current sampler name and replaces it with the name pointed to by samplerArg.

Syntax

```
int SetSampler( const char *samplerArg )
```

Parameters

Input

samplerArg–A pointer to a string that holds the sampler name

Output

An integer value: 0 indicates success; -1 indicates failure

SetTextDescription method

This method deletes the current text description and replaces it with the given string. The text description is specific to the text query for QBIC.

Syntax

```
int SetTextDescription ( const char *text )
```

Parameters

Input

text–A pointer to a string that contains the text description for the query, such as a set of keywords

Output

An integer value: 0 indicates success; -1 indicates failure

SetType method

This method sets the type of data source in QbGenericDataClass.

Syntax

```
void SetType( const dataSource typeArg )
```

Parameters

Input

typeArg–An enumerated type of dataSource described on page 111

Output

None

ToQueryString method

This method translates generic and domain-specific information inside the object to a byte string.

Syntax

```
virtual int ToQueryString( QbDatumClass &datum )
```

Parameters

Input

datum–A reference to a QbDatumClass object that will hold the byte string of the encoded data

Output

An integer value: 0 indicates success; -1 indicates failure

Type method

This method returns a constant string of the class name.

Syntax

```
virtual const char * Type( void )
```

Parameters

Input

None

Output

A constant string, QbGenericDataClass

QbGenericImageDataClass 15

QbGenericImageDataClass is derived from the QbGenericDataClass. It includes all the methods of the superclass, plus other methods specific to processing images, such as reading the image from memory or disk, getting the image's size, and specifying a mask image which allow you to query a subpart of an image based on a reference image (the mask).

ToQueryString in the derived class is implemented to encode the domain-specific data, and then to call the base class ToQueryString method to encode the generic information. FromQueryString reverses this process.

QbGenericImageDataClass includes methods ReadImageDataFromFile and ReadImageDataFromPickerFile, which perform image I/O. They support reading a variety of common image formats, including:

- OS/2 and Windows bitmap image (bmp) (BA-type only)
- GIF 87 or 89a (gif)
- JPEG (jpg/jpeg)
- PBMPLUS: portable graymap (pgm)
- PBMPLUS: portable pixmap (ppm)
- TIFF 5.0 image (tiff/tif), including CCITT Group 3 and Group 4 fax image
- Targa (tga)

In addition to the standard image formats listed above, the QbGenericImageDataClass can also read a QBIC Picker Image Description string either in memory (ReadPickerImageDescriptionString) or in a file (ReadPickerImageDescriptionFile). This description string is a simple encoding of an image in a text string. Currently, this method only recognizes rectangles. The string format is:

```
Dwidth,height:Rulx,uly,rwidth,rheight,R,G,B:...
```

where:

- D specifies the Drawing area with the dimensions width and height. The origin is in the upper left corner.
- : (colon) is the field separator
- R specifies a Rectangle with the upper left corner at ulx,uly, relative to the drawing area's origin. Dimensions are rwidth, rheight. Color is specified in RGB format as R,G,B.

You can specify multiple rectangles (R strings), but the first rectangle must be “painted” in the drawing area first, and subsequent rectangles which overlap are painted over it. QBIC considers any area that is not painted is “to be ignored”. Color information inside the “to be ignored” region does not affect the QBIC distance evaluation.

The QBIC Picker Image Description string is useful for sending image descriptions from a client to a server so that the server can construct a query image based on the description, and then search for similar images in the database. The description string is heavily used in the QBIC demo.

This chapter describes the following methods:

You can see coding examples in Appendix C, “Sample Code” on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbGenericImageDataClass method

This is the constructor for the class.

Syntax

```
QbGenericImageDataClass( void )
```

Parameters

None

~QbGenericImageDataClass method

This is the destructor for the class.

Syntax

```
virtual ~QbGenericImageDataClass( void )
```

Parameters

None

DecodeSubPartDef method

This method decodes a subpart definition from a QbDatumClass instance, and initializes the object's private variables describing the subparts. This method assumes the image that corresponds to the parent key has already been stored in the object. If not, this method will give unexpected results.

Syntax

```
virtual int DecodeSubPartDef( const QbDatumClass &subPartDatum )
```

Parameters

Input

subPartDatum–A reference to a QbDatumClass object whose byte string holds the subpart information to be decoded

Output

An integer value: 0 indicates success; -1 indicates failure

EncodeSubPartDef method

This method encodes the subpart definition inside the object to a byte string, and assigns it to the byte string of the input QbDatumClass object.

Syntax

```
virtual int EncodeSubPartDef( QbDatumClass &subPartDatum )
```

Parameters

Input

subPartDatum–A reference to a QbDatumClass object that will hold the encoded subpart description

Output

An integer value: 0 indicates success; -1 indicates failure

FromQueryString method

This method decodes a byte string of a QbDatumClass object to initialize the object.

Syntax

```
virtual int FromQueryString( const QbDatumClass &datum, const char
    *featureClassname )
```

Parameters

Input

- **datum**–A reference to a QbDatumClass object that holds the encoded generic and image-specific data
- **featureClassname**–A pointer to a string that holds the feature class name, such as QbColorHistogramClass

Output

An integer value: 0 indicates success; -1 indicates failure

GetFilename method

This method returns the image name whose data the current object represents.

Syntax

```
const char * GetFilename( void )
```

Parameters

Input

None

Output

A pointer to a string to hold the file name

GetImage method

This method returns a pointer to an array of image pixel indexing.

Syntax

```
unsigned char * GetImage( void )
```

Parameters

Input

None

Output

A pointer to an unsigned char. Its size is Xsize*Ysize, where Xsize and Ysize are the width and height of the image. The pixel index value at point (x,y) should be:

```
GetImage()[y*Xsize+x]
```

For more information, see the “GetLut method” on page 131, “GetXsize method” on page 132, and “GetYsize method” on page 132. Also, see the “GetImageDataFromMemory method” on page 130.

GetImageDataFromMemory method

This method initializes a QbGenericImageDataClass object using a given image in memory. The memory image should include a palette with a color table or look-up table.

Syntax

```
int GetImageDataFromMemory( const unsigned long width, const unsigned
    long height, unsigned char *imageArg, unsigned char *lutArg,
    unsigned char *maskArg, const Boolean subpart = False, Boolean
    deletesStorageArg = False )
```

Parameters

Input

- **width**–An unsigned long integer that represents the image's width in pixels
- **height**–An unsigned long integer that represents the image's height in pixels
- **imageArg**–A pointer to an unsigned char that is an array of size width * height that stores the pixel index values. The pixel index value p at x and y would be p = imageArg[y * width + x].
- **lutArg**–A pointer to an unsigned char that is an array of dimension 786 that stores the RGB values for each index value. The RGB value for an index value should be lutArg[p], lutArg[p+256], lutArg[p+512], respectively.
- **maskArg**–A pointer to an unsigned char array that stores the image masks for each subpart, which can be NULL
- **subpart**–A Boolean value that indicates whether the description is for a subpart image; the default is False. If subpart is set to True, maskArg cannot be a NULL. Only the calling function can determine whether or not this object should be a subpart.
- **deletesStorageArg**–A Boolean value that indicates whether the class destructor should delete storage passed to the object, such as imageArg, lutArg, and maskArg. The default is False. If deletesStorageArg = True, it is assumed that the storage was created by QbNew or QbCNew. See "QbNew method" on page 167 and "QbCNew method" on page 166 for details.

Output

An integer value: 0 indicates success; -1 indicates failure

GetLut method

This method returns a pointer to the look-up table, which has an array size of 768 (256 * 3).

Syntax

```
unsigned char * GetLut( void )
```

Parameters

Input

None

Output

A pointer to an unsigned char array

GetMask method

This method returns a pointer to the array to hold the image mask.

Syntax

```
unsigned char * GetMask( void )
```

Parameters

Input

None

Output

A pointer to an unsigned char array that holds the image mask

GetMaskname method

This method returns the mask image name whose data the current object represents.

Syntax

```
const char * GetMaskname( void )
```

Parameters

Input

None

Output

A pointer to a string to hold the mask file name

GetXsize method

This method returns the width of the image.

Syntax

```
unsigned long GetXsize( void )
```

Parameters

Input

None

Output

An unsigned long integer to hold the image’s width

GetYsize method

This method returns the height of the image.

Syntax

```
unsigned long GetYsize( void )
```

Parameters

Input

None

Output

An unsigned long integer to hold the image’s height

Itype method

This method returns the interface type defined in the base class.

Syntax

```
virtual const QbInterfaceType Itype( void )
```

Parameters

Input

None

Output

GenericDataInterfaceType of QbInterfaceType defined in the QbBaseClass. See page 49.

ReadImageDataFromFile method

This method reads in image data from a file.

Syntax

```
int ReadImageDataFromFile( const char *filename, const char
    *maskFileName = NULL, const Boolean subpart = False )
```

Parameters

Input

- **filename**–A pointer to a string that contains the file name
- **maskFileName**–A pointer to a string that contains the mask file name; can be NULL
- **subpart**–A Boolean value that indicates whether the description is for a subpart image; the default is False. If this value is True, then the maskFileName value cannot be NULL

Output

An integer value: 0 indicates success; -1 indicates failure

ReadImageDataFromPickerFile method

This method reads in image data from a picker file.

Syntax

```
int ReadImageDataFromPickerFile( const char *filename, const char
    *maskFileName = NULL, const Boolean subpart = False )
```

Parameters

Input

- **filename**–A pointer to a string that contains the picker file name
- **maskFileName**–A pointer to a string that contains the mask file name; can be NULL
- **subpart**–A Boolean value that indicates whether the description is for a subpart image; the default is False. If this value is True, then the maskFileName value cannot be NULL

Output

An integer value: 0 indicates success; -1 indicates failure

ReadPickerImageDescriptionFile method

This method reads a picker image description from a file.

Syntax

```
int ReadPickerImageDescriptionFile( const char *filename, const
    Boolean subpart = False )
```

Parameters

Input

- **filename**–A pointer to a string that contains the picker file name
- **subpart**–A Boolean value that indicates whether the description is for a subpart image; the default is False

Output

An integer value: 0 indicates success; -1 indicates failure

ReadPickerImageDescriptionString method

This method reads a picker image description from a string. QBIC currently supports only rectangles for this method.

Syntax

```
int ReadPickerImageDescriptionString( const char *dstr, const Boolean
    subpart = False )
```

Parameters

Input

- **dstr**–A pointer to a string that contains the picker description
- **subpart**–A Boolean value that indicates whether the description is for a subpart image; the default is False

Output

An integer value: 0 indicates success; -1 indicates failure

ToQueryString method

This method encodes the content of the object to a byte string and assigns it to the byte string of the input QbDatumClass object.

Syntax

```
virtual int ToQueryString( QbDatumClass &datum )
```

Parameters

Input

datum–A reference to a QbDatumClass object that will hold the byte string that encodes the object data

Output

An integer value: 0 indicates success; -1 indicates failure

Type method

This method returns a constant string of the class name.

Syntax

```
virtual const char * Type( void )
```

Parameters

Input

None

Output

A constant string, QbGenericImageDataClass

UpdateMaskInImageObject method

This method reads the given maskFileName and updates the object's image mask.

Syntax

```
int UpdateMaskInImageObject( const char *maskFileName )
```

Parameters

Input

maskFileName–A pointer to a string that contains the mask file name

Output

An integer value: 0 indicates success; -1 indicates failure

QbKeyDatabaseClass 16

QbKeyDatabaseClass is derived from the QbDatabaseClass and returns the results of queries and specifies a restriction list of keys. A restriction list is a list of keys used to restrict a query. For example, in a query such as “Find images by artist Picasso with this shade of red” with a text query on “Picasso” returns a set of keys of images associated with Picasso. These keys are then passed to the query engine as a restriction list together with the content-based query specification for “red” to obtain the final, ordered query results.

QbKeyDatabaseClass is a database of keys with distance values. Iterations on this database are required to return items in increasing distance order. Key order is used in the event of a tie.

The IsRanked and SetRanked methods allow you to find or set the ranked state of this database. Even though the database is always sorted by distance, this sort may not be valid.

The GetDistance and SetDistance methods return or set the distance of a value datum. When you use a QbKeyDatabaseClass instance to define a restriction list, use the same distance for all elements or call SetDistance with no distance argument (it defaults to use the distance of -1.0).

The GetNumberOfKeys method returns the number of keys (records) in the database. The return results of a query are always ranked.

The RetrieveMax method retrieves the element with the largest distance in the list. An efficient implementation of this method could greatly enhance query performance.

To pass restriction lists to a query, use the ToQueryString method. This process can be reversed using the FromQueryString method. In the current implementation, the ToQueryString method creates a blank terminated list of key distance pairs, and FromQueryString reinserts these into a QbKeyDatabaseClass object.

This chapter describes the following methods:

You can see coding examples in Appendix C, “Sample Code” on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbKeyDatabaseClass method

This is the constructor for the class.

Syntax

```
QbKeyDatabaseClass( void )
```

Parameters

None

~QbKeyDatabaseClass method

This is the destructor for the class.

Syntax

```
virtual ~QbKeyDatabaseClass( void )
```

Parameters

None

FromKeyString method

This method decodes the string inside a QbDatumClass object to initialize the object. Distance values for all keys are set to zero.

Syntax

```
virtual int FromKeyString( const QbDatumClass &datum )
```

Parameters

Input

datum–A reference to a QbDatumClass object whose byte string holds names of keys separated by spaces

Output

An integer value: 0 indicates success; -1 indicates failure

FromQueryString method

This method decodes the byte string in the input QbDatumClass object and initializes the QbKeyDatabaseClass object.

Syntax

```
virtual int FromQueryString( const QbDatumClass &datum )
```

Parameters

Input

datum–A reference to a QbDatumClass object that holds the encoded key-distance pairs

Output

An integer value: 0 indicates success; -1 indicates failure

GetDistance method

This method retrieves the distance value from the input QbDatumClass object.

Syntax

```
virtual float GetDistance( const QbDatumClass &data )
```

Parameters

Input

data–A reference to a QbDatumClass object whose byte string holds the distance value

Output

A float value that represents the distance

GetNumberOfKeys method

This method returns the number of elements in the object.

Syntax

```
unsigned long GetNumberOfKeys( void )
```

Parameters

Input

None

Output

An unsigned long integer that represents the number of elements

IsRanked method

This method returns True if the database is set to order the keys based on the distance value, False if not.

Syntax

```
Boolean IsRanked( void )
```

Parameters

Input

None

Output

A Boolean value; True if the database is set to order the keys based on the distance value, False if it is not.

RetrieveMax method

This method retrieves the key-value pair that contains the maximum distance in the object.

Syntax

```
virtual int RetrieveMax( QbDatumClass &key, QbDatumClass &value )
```

Parameters

Input

- **key**–A reference to a QbDatumClass object that holds the key
- **value**–A reference to a QbDatumClass object that holds the value

Output

An integer value: 0 indicates success; 1 indicates nonfatal error (for example, no keys in object); -1 indicates fatal error

SetDistance method

This method assigns the distance value into the byte string in the input data object.

Syntax

```
virtual int SetDistance( QbDatumClass &data, const float distance = -1
    )
```

Parameters

Input

- **data**–A reference to a QbDatumClass object that will hold the distance value
- **distance**–A float that represents the distance

Output

An integer value: 0 indicates success; -1 indicates failure

SetRanked method

This method makes the object order keys based on the distance value if rank=True.

Syntax

```
void SetRanked( const Boolean rank )
```

Parameters

Input

rank–A Boolean value; True means rank on distance, False means do not.

Output

None

ToKeyString method

This method encodes keys inside an object to an ASCII string and assigns it to the byte string of the given QbDatumClass object. The distance records are not encoded.

Syntax

```
virtual int ToKeyString( QbDatumClass &datum )
```

Parameters

Input

datum–A reference to a QbDatumClass object whose byte string will hold the encoded keys

Output

An integer value: 0 indicates success; -1 indicates failure

ToQueryString method

This method encodes both key and distance values in the object to an ASCII string and assigns it to the a byte string in the input QbDatumClass object.

Syntax

```
virtual int ToQueryString( QbDatumClass &datum )
```

Parameters

Input

datum–A reference to a QbDatumClass object whose byte string will hold encoded key-distance pairs

Output

An integer value: 0 indicates success; -1 indicates failure

QbParameterClass 17

QbParameterClass is derived from the QbBaseClass and encapsulates the parameters associated with a QBIC feature computation or QBIC query. It is fully implemented, and creates an array of name/value pairs that any derived class of QbFeatureClass can use.

Both the QbParameterClass and the QbGenericDataClass define the methods ToQueryString and FromQueryString. These methods convert the class to a byte string, which allows the query to be passed from a client to a server.

Each QBIC feature class constructs its own default QbParameterClass and initializes it with appropriate settings for the feature class. You can fine tune it by passing your own QbParameterClass pointer during feature computation. During the query, you can pass parameters to each feature class to determine the class weights and query orders in multi-feature and multi-pass queries.

This chapter describes the following methods:

You can see coding examples in Appendix C, "Sample Code" on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbParameterClass method

This is the constructor for the class.

Syntax

```
QbParameterClass( void )
```

Parameters

None

~QbParameterClass method

This is the destructor for the class.

Syntax

```
virtual ~QbParameterClass( void )
```

Parameters

None

FromCommandLineString method

This method uses one of the following overloaded functions to initialize an object:

- A string description of the parameters, or
- A byte string (from a datum object). The byte string in the datum object has a slightly different format from the corresponding one used in the FromQueryString method, so the byte string is enclosed in brackets (< >).

Syntax

```
int FromCommandLineString( const char *parmString )

int FromCommandLineString( const QbDatumClass &datum )
```

Parameters

Input

- **parmString**–For the first function, a pointer to a string containing a description of the parameters
- **datum**–For the second function, a reference to a QbDatumClass object that holds a byte string containing a description of the parameters

Output

An integer value; 0 indicates success; -1 indicates failure

FromQueryString method

This method has overloaded functions. This method decodes one of the following to initialize an object, depending on the function used: a string description of the parameters or a byte string (from a datum object).

Syntax

```
int FromQueryString( const char *string )

int FromQueryString( const QbDatumClass &datum )
```

Parameters

Input

- **string**–For the first function, a pointer to a string that holds a description of the parameters
- **datum**–For the second function, a reference to a QbDatumClass object whose byte string holds a description of the parameters

Output

An integer value; 0 indicates success; -1 indicates failure

FromQueryStringInFile method

This method reads the string from the named file and decodes the string to initialize the object.

Syntax

```
int FromQueryStringInFile( const char *filename )
```

Parameters

Input

filename–A pointer to a string that holds the file name

Output

An integer value; 0 indicates success; -1 indicates failure

GetClassWeight method

This method returns the class weight of the object.

Syntax

```
fastfloat GetClassWeight( void )
```

Parameters

Input

None

Output

A fastfloat (double) value representing the class weight for multi-feature or multi-pass queries

GetCombineFunc method

This method returns an integer that indicates how the distances in a multi-feature or multi-pass query should be combined. If it is 0, the distances from different features at different passes are added (after multiplying by the appropriate class weights). If it is 1, the distance of the last query pass is used and all previously passed distances are discarded (after multiplying by the appropriate class weights) instead.

Syntax

```
int GetCombineFunc( void )
```

Parameters

Input

None

Output

An integer value, 0 or 1

GetCount method

This method returns the number of feature codes in the current object. Each QBIC feature class has its own set of feature codes which can be set by using the SetFeatureCode method.

Syntax

```
unsigned int GetCount const( void )
```

Parameters

Input

None

Output

An unsigned integer value representing the number of feature codes currently stored in the object

GetFeatureCodeNames method

This method returns all of the feature code names for the object.

Syntax

```
char ** GetFeatureCodeNames( void )
```

Parameters

Input

None

Output

A string array with a count of elements (get the count by using the GetCount method). For any number i < count, GetFeatureCodeNames()[i] points to a string describing the i-th feature code name.

GetFeatureCodes method

This method returns feature codes for an object. This method has overloaded functions. Depending on the function, it either returns all of the feature code values for the object or returns the feature code that corresponds to the given feature code name. The feature code values are used to fine tune the way a QBIC feature class performs feature computation. Different feature classes have different sets of feature code values.

Syntax

```
char ** GetFeatureCodes( void )

char * GetFeatureCodes( const char *name )
```

Parameters

Input

- For the first function, none
- **name**–For the second function, a pointer to a string that contains a feature code name

Output

- For the first function, a string array with a count of elements (get the count by using the GetCount method). For any number i < count, GetFeatureCodes()[i] points to a string describing the i-th feature code value.
- For the second function, a string that contains the feature code value for the given feature code name

GetFilterFlag method

If this returns True, the object instructs QBIC to use the QbFeatureClass' index scheme to increase the speed of the query.

Syntax

```
Boolean GetFilterFlag( void )
```

Parameters

Input

None

Output

A Boolean value; True means use the QbFeatureClass' index scheme, False means do not.

GetQueryOrder method

This method returns the order in which multi-pass QBIC queries are made. The default value is 0, which represents the highest priority.

Syntax

```
int GetQueryOrder( void )
```

Parameters

Input

None

Output

An integer that represents the order of a QBIC query.

GetWeights method

This method returns feature weights. This method has overloaded functions. Depending on the function, it either returns a pointer to the feature weight array or the feature weight for the given feature code name.

Syntax

```
fastfloat * GetWeights( void )
```

```
fastfloat GetWeights( const char *name )
```

Parameters

Input

- For the first function, none

- **name**–For the second function, a pointer to a string that contains a feature code name

Output

- For the first function, a pointer to a fastfloat (double) array with the count of elements (get the count using the GetCount() method). For any number i < count, GetWeights()[i] points to a fastfloat describing the weight for the i-th feature code name. All weights default to 0.
- For the second function, a fastfloat (double) that represents the feature weight for the given feature code name.

Itype method

This method returns the interface type defined in the base class.

Syntax

```
virtual const QbInterfaceType Itype( void )
```

Parameters

Input

None

Output

ParameterInterfaceType of QbInterfaceType defined in the QbBaseClass. See page 49.

SetClassWeight method

This method sets the value of the class weight for the object.

Syntax

```
void SetClassWeight( fastfloat x )
```

Parameters

Input

x–A fastfloat (double) value representing the class weight for multi-feature or multi-pass queries. The default is 1.

Output

None

SetCombineFunc method

This method sets the way the distances in a multi-pass query should be combined.

Syntax

```
void SetCombineFunc( int x )
```

Parameters

Input

x–An integer value, 0 or 1. Set it to 0 to add distances from different features at different passes (after multiplying by the appropriate class weights). Set it to 1 to use the distance of the last query pass and discard distance values of all previous passes (after multiplying by the appropriate class weights).

Output

None

SetFeatureCodes method

This method adds or replaces a feature code value for the corresponding feature code name.

Syntax

```
int SetFeatureCodes( const char *name, const char *value )
```

Parameters

Input

- **name**–A pointer to a string that contains the feature code name
- **value**–A pointer to a string containing the feature code value

Output

An integer value: 0 indicates success; -1 indicates failure

SetFilterFlag method

This method sets the object such that if the value is True and an index scheme is implemented, QbFeatureClass' FilterResult method is called during the query to increase the speed of the query.

Syntax

```
void SetFilterFlag( const Boolean value )
```

Parameters

Input

A Boolean value; True means the FilterResult method is called, False means it is not

Output

None

SetQueryOrder method

This method sets the order for a multi-pass query.

Syntax

```
void SetQueryOrder( int x )
```

Parameters

Input

x–An integer that represents the order of a QBIC query

Output

None

SetWeights method

This method adds or replaces the feature code weight for a given feature code name.

Syntax

```
int SetWeights ( const char *name, const fastfloat value )
```

Parameters

Input

- **name**–A pointer to a string that contains a feature code name
- **value**–A fastfloat (double) that represents the new feature code weight

Output

An integer value: 0 indicates success; -1 indicates failure

ToCommandLineString method

This method encodes data inside the object into a byte string and assigns it to the byte string of datum.

Syntax

```
int ToCommandLineString( QbDatumClass &datum )
```

Parameters

Input

datum–A reference to a QbDatumClass object whose byte string will hold the encoded description of the object

Output

An integer value; 0 indicates success; -1 indicates failure

ToQueryString method

This method encodes the object parameters into a byte string and assigns it to a byte string (in a datum object).

Syntax

```
int ToQueryString( QbDatumClass &datum )
```

Parameters

Input

datum–A reference to a QbDatumClass object to hold the encoded byte string containing a description of the parameters

Output

An integer value; 0 indicates success; -1 indicates failure

Type method

This method returns a constant string of the class name.

Syntax

```
virtual const char * Type( void )
```

Parameters

Input

None

Output

A constant string, QbParameterClass

QbQueryClass

18

QbQueryClass is derived from the QbBaseClass and is an abstract class that defines the Evaluate method. The input to the Evaluate method is the query string, which is a simple text based query language consisting of text strings that define a content-based query. The query string is built using the QbBuildQueryString method defined in QbQueryS.hpp.

QbBuildQueryString arguments include the number of results requested, catalog name, optional restriction lists, number of features, feature class names, and a parameter class and generic data pointer for each feature. It returns the query string in a QbDatum object. Use the MakeQueryByExampleObject and MakeQueryByTextObject methods to make a valid QbGenericDataClass object for this call.

The QbQueryServerClass derived class has several private methods to decode the query string, optimize, prepare, and execute the query. Using the query string allows you to separate the functionality of client and server, when they are in different process spaces or on different platforms. If they are co-located, you can define an overloaded version of the Evaluate method that takes several class pointers to fulfill the same functionality.

This chapter describes the following methods:

You can see coding examples in Appendix C, "Sample Code" on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbQueryClass method

This is the constructor for the class.

Syntax

```
QbQueryClass( void )
```

Parameters

None

~QbQueryClass method

This is the destructor for the class.

Syntax

```
virtual ~QbQueryClass( void )
```

Parameters

None

Evaluate method

This method has an overloaded abstract interface for defining a QBIC query.

Syntax

```
virtual const QbKeyDatabaseClass * Evaluate( const char *query )

virtual const QbKeyDatabaseClass * Evaluate( const int
    numberOfResults, QbCatalogClass *cat, QbKeyDatabaseClass *res,
    QbKeyDatabaseClass *parts, int numOfFeatures, QbGenericDataClass
    **data, QbFeatureClass **feat, QbParameterClass **parm )
```

Parameters

Input

- **query**–For the first function, a pointer to a character string that contains the query specification that was built using QbBuildQuery String defined in QbQueryS.hpp
- **numberOfResults**–For the second function, an integer that defines how many keys you want QBIC to return in the output. The actual number of returned keys can be smaller than numberOfResults if there are not enough keys to meet the search criteria.
- **cat**–For the second function, a pointer to a QbCatalogClass object. This pointer cannot be NULL and the object must have been opened for a QBIC query.
- **res**–For the second function, a const pointer to a QbKeyDatabaseClass object that is used to restrict the search keys contained in the object. This pointer can be NULL; if it is NULL, QBIC searches the whole database.
- **parts**–For the second function, a const pointer to a QbKeyDatabaseClass object that is used to restrict the search when the feature contains a subpart. This pointer can be NULL.

- **numberOfFeatures**–For the second function, an integer that defines how many features you want QBIC to use in the search. This number must be greater than 0. If it is greater than 1, the search corresponds to a multi-pass or multi-feature query.
- **data**–For the second function, a pointer array to QbGenericDataClass objects. A number of pointers equal to the numberOfFeatures value must not be NULL, and they must describe the input feature data for the query.
- **feat**–For the second function, a pointer array to QbFeatureClass objects. A number of pointers equal to the numberOfFeatures value must not be NULL, and they must describe the features to search for.
- **parms**–For the second function, a pointer array to QbParameterClass objects. A number of pointers equal to the numberOfFeatures value must not be NULL, and they must describe the parameters for the corresponding QBIC Feature Class that gives the class weight, search order, and so on.

Output

A pointer to a QbKeyDatabaseClass object that maintains a ranked list of key and distance pairs for QBIC query results. You can build the query string using the QbBuildQueryString method defined in QbQueryS.hpp. It builds a QbDatumClass object from which you can retrieve the string using its Get method.

GetReturnCode method

This method retrieves the return code for the Evaluate method.

Syntax

```
virtual int GetReturnCode( void )
```

Parameters

Input

None

Output

An integer value: 0 indicates success; -1 indicates failure

QbStringClass 19

QbStringClass is a utility class that is not derived from any other class. It performs string functions and related memory management. The class contains a set of overloaded operators to allow the object to be cast to either an unsigned char or char variable type.

In this class, the += operator has been overloaded to concatenate two QbString objects.

This chapter describes the following methods:

You can see coding examples in Appendix C, "Sample Code" on page 185. Other programs are located in the `qbic/QbicApi` directory.

QbStringClass method

This is the constructor for the class. This method has overloaded functions. Depending on which function you use, the length of the object's internal buffer is the same as the length of the input string, or is a size you specify. The second argument indicates that the memory of the internal buffer is cleared when the object is destroyed.

Syntax

```
QbStringClass( const char * string, Boolean destructorDeleteArg = True
    )

QbStringClass( const unsigned long size, Boolean destructorDeleteArg =
    True )
```

Parameters

Input

- **string**–For the first function, a pointer to a character string to be initialized

- **size**–For the second function, an unsigned long integer that represents the size of the object's internal buffer
- **destructorDeleteArg**–A Boolean that represents whether the internal buffer is to be cleared when the object is destroyed; it is set to True by default

Output

None

~QbStringClass method

This is the destructor for the class.

Syntax

```
virtual ~QbStringClass( void )
```

Parameters

None

EncodeAndConcat method

This method encodes the input string to a hexadecimal representation and appends the result to the internal buffer of the object.

Syntax

```
int EncodeAndConcat ( const unsigned char *input, const unsigned long
   size )
```

Parameters

Input

- **input**–A pointer to an unsigned char that represents the string to be encoded
- **size**–An unsigned long integer that represents the size of the input string

Output

An integer value: 0 indicates success; -1 indicates failure

GetSize method

This method returns the size of the object's internal buffer that holds the string.

Syntax

```
unsigned long GetSize( void )
```

Parameters

Input

None

Output

An unsigned long integer that represents the size of the object's internal buffer

IsOk method

This method returns a Boolean value that indicates the current state of the object. Any operation that requires memory, such as the += operator, could fail and leave the object in an "unhealthy" state.

Syntax

```
Boolean IsOK( void )
```

Parameters

Input

None

Output

A Boolean value: True means the object is "healthy"; False means the object experienced a memory allocation error

SetDestructorDelete method

This method indicates whether the destructor should delete the internal buffer. This method overrides what you specify in the class constructor for destructorDeleteArg.

Syntax

```
void SetDestructorDelete( Boolean destructorDeleteArg )
```

Parameters

Input

destructorDeleteArg–A Boolean value: True indicates that the destructor will delete the buffer; False indicates that it will not.

Output

None

QBIC Error Handling Routines

20

To use QBIC's error handling routines, call QbInitializeErrorStructure(), which is declared in quError.h, at the beginning of your program or before any QBIC calls.

When an error is detected (as indicated by a nonzero return code from a class method or when False is returned after invoking the IsOK() method), you can retrieve the error severity state by calling GetErrorMSGSeverity().

If the severity state is QERR_FATAL, the system is in a fatal error state and should exit gracefully. To determine the problem, call QbPrintErrorMessage(FILE *stream) to print the error message to the specified IO stream.

Because most errors happen during memory allocation, the QBIC API provides the following set of routines specifically designed for memory allocation. Use these methods if you want to write your own feature extraction routines.

The following methods are described in this chapter:

GetErrorMSGSeverity method

This method returns error severity information.

Syntax

```
QERR_SeverityType GetErrorMSGSeverity(void);
```

Parameters

Input

None

Output

A enumerated value of either QERR_NON_FATAL or QERR_FATAL.

QbCNew method

This method is similar to QbNew, except that it creates memory for a specified number of objects as (nobj * size).

Syntax

```
void *QbCNew(size_t nobj, size_t size, const char *err_msg,
   QERR_SeverityType Sev = QERR_FATAL)
```

Parameters

Input

- **nobj**–Number of objects to allocate memory for
- **size**–Size of each object, in bytes
- **err_msg**–Pointer to the error message
- **Sev**–Enumerated type that indicates the error severity level. The error Severity level defaults to QERR_FATAL. You can override it by setting it to QERR_NON_FATAL.

Output

None

QbDelete method

This method frees memory allocated by the QbNew and QbCNew methods.

Syntax

```
void QbDelete(void *p)
```

Parameters

Input

p–Pointer to the memory block to be freed

Output

None

QbInitializeErrorStructure method

This method initializes the error handling routine.

Syntax

```
QbInitializeErrorStructure();
```

Parameters

Input

None

Output

None

QbNew method

This method allocates a block of memory. If it fails, it returns a NULL and fills the QBIC error message buffer with the message (pointed to by the string err_msg).

Syntax

```
void *QbNew(size_t size,  const char *err_msg , QERR_SeverityType Sev =
   QERR_FATAL)
```

Parameters

Input

- **size**–Size of the memory block, in bytes
- **err_msg**–Pointer to the error message
- **Sev**–Enumerated type that indicates the error severity level. The error Severity level defaults to QERR_FATAL. You can override it by setting it to QERR_NON_FATAL.

Output

None

QbPrintErrorMessage method

This method prints the error message to the specified file descriptor.

Syntax

```
void QbPrintErrorMessage(FILE *stream)
```

Parameters

Input

- **stream**–File descriptor to which the error message will be printed.

Output

None

QBIC-Specific tcl Function Calls

A

qbictcl extends the tcl script with QBIC functionality. The following QBIC-specific function calls are available in qbictcl (in addition to standard tcl function calls):

get_host_ip

Locates the host computer's IP address. This function call is not available on the Macintosh.

Syntax

```
get_host_ip
```

Parameters

None

Example

```
set ip [get_host_ip]
```

qbic_add_feature

Adds the specified feature to the qbic descriptor.

Syntax

```
qbic_add_feature qbicDescriptor FeatureName
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **FeatureName**–Feature to be added

Example

```
qbic_add_feature $qb QbColorHistogramFeatureClass
```

qbic_add_image

Computes and stores feature data for the input image. Function calls to qbic_add_feature function must have been called, and only those features that were described in the qbic_add_feature call will be computed. The image name is used as the key.

Syntax

```
qbic_add_image qbicDescriptor imageName
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **imageName**–Image to be added

Example

```
qbic_add_image $qb flower01.jpg
```

qbic_clock

Gets the current clock time, in milliseconds, and can be used for the purpose of timing performance.

Example

```
set t1 [qbic_clock]

   ....

set t2 [qbic_clock]

puts stdout "time spent: [expr $t2 - $t1] msec"
```

qbic_delete_feature

Deletes the specified feature from the qbic descriptor.

Syntax

```
qbic_delete_feature qbicDescriptor FeatureName
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **FeatureName**–Feature to be deleted

Example

```
qbic_delete_feature $qb QbColorHistogramFeatureClass
```

qbic_delete_image

Deletes a specified image key from the qbic catalog.

Syntax

```
qbic_delete_image qbicDescriptor imageKey
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **imageName**–Image key to be deleted

Example

```
qbic_delete_image $qb flower01.jpg
```

qbic_end

Closes QBIC, and deletes any QBIC objects from memory.

Syntax

```
qbic_end qbicDescriptor
```

Parameters

qbicDescriptor–Descriptor returned by qbic_start

Example

```
qbic_end $qb
```

qbic_get_param

Gets the parameter for the named feature class.

Syntax

```
qbic_get_param featureName ?qbicDescripter?
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **featureName**–Feature name for the desired parameter

Example

```
qbic_get_param QbColorHistogramFeatureClass
```

qbic_image_fdata

Gets the feature data for the input image name.

Syntax

```
qbic_image_fdata qbicDescriptor imageName
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **imageName**–Input image name for the feature data to be computed

Example

```
qbic_image_fdata $qb flower01.jpg
```

qbic_key_fdata

Gets the feature data for the input image key.

Syntax

```
qbic_key_fdata qbicDescriptor keyName
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **keyName**–Key name for the feature data to be retrieved

Example

```
qbic_key_fdata $qb flower01.jpg
```

qbic_list_all_features

Lists all QBIC features.

Syntax

```
qbic_list_all_features
```

Parameters

None

Example

```
set allfeatures [qbic_list_all_features]
```

qbic_list_cat_features

Lists all QBIC features for a particular catalog.

Syntax

```
qbic_list_cat_features qbicDescriptor
```

Parameters

qbicDescriptor–Descriptor returned by qbic_start

Example

```
qbic_list_cat_features $qb
```

qbic_make_connect

Closes any QBIC catalogs (if opened), and connects to the specified catalog and database using the specified connection mode.

Syntax

```
qbic_make_connect qbicDescriptor catalogName, databaseName,
   connectionMode
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **catalogName**–Specify a catalog name to connect to
- **databaseName**–Specify a database name to connect to
- **connectionMode**–Specify a connection mode (r for read; w for write)

Example

```
qbic_make_connect $qb ibm33 qbic/QbicData r
```

qbic_make_thumb

Creates a thumbnail from a specified input image.

Syntax

```
qbic_make_thumb inImage outImage
```

Parameter

- **inImage**–Image for which you want to create a thumbnail
- **outImage**–Image name of thumbnail

Example

```
qbic_make_thumb flower01.jpg flower01.thumb.jpg
```

qbic_picker_fdata

Gets the feature data for the input picker image key.

Syntax

```
qbic_key_fdata qbicDescriptor keyName
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **keyName**–Picker key n.Name for the feature data to be retrieved.

Example

```
qbic_picker_fdata $qb flower01.jpg
```

qbic_query_image

Queries the database of the given image for similar ones.

Syntax

```
qbic_query_image qbicDescriptor imageName
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **imageName**–Image name used for query

Example

```
qbic_query_image $qb flower01.jpg
```

qbic_query_key

Queries a key (image feature data are already in the QBIC database) for similar ones.

Syntax

```
qbic_query_key qbicDescriptor keyName
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **keyName**–Image specified for query

Example

```
qbic_query_key $qb flower01.jpg
```

qbic_set_keys_return

Sets how many query keys will be returned for the query.

Syntax

```
qbic_set_keys_return qbicDescriptor num
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **num**–Number of keys to return

Example

```
qbic_set_keys_return $qb 20
```

qbic_set_keyword

Sets the keyword for a QBIC keyword search.

Syntax

```
qbic_set_keyword qbicDescriptor
```

Parameters

qbicDescriptor–Descriptor returned by qbic_start

Example

```
qbic_set_keyword $qb "flower"
```

qbic_set_thumb_24color

Sets QBIC to use 24-bit color (full color) to generate thumbnail images.

Syntax

```
qbic_set_thumb_24color qbicDescriptor
```

Parameters

qbicDescriptor–Descriptor returned by qbic_start

Example

```
qbic_set_thumb_24color $qb
```

qbic_set_thumb_size

Specifies the thumbnail size generated by QBIC.

Syntax

```
qbic_set_thumb_size qbicDescriptor width height
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **width**–Specifies the width of the thumbnail in pixels
- **height**–Specifies the height of the thumbnail in pixels

Example

```
qbic_set_thumb_size $qb 100 100
```

qbic_start

Starts QBIC. Will return a qbic descriptor, which should be used in other QBIC function calls, for example, in the qbic_add_feature call.

Syntax

```
qbic_start ? catName, databaseName, mode?
```

Parameters

- **catName**–Specify a catalog name
- **databaseName**–Specify a database name
- **mode**–Specify a connection mode, either “r” or “w”

Example

```
set qb [qbic_start ibm33 qbic/QbicData r]
```

qbic_string_fdata

Gets the feature data for the input image description string.

Syntax

```
qbic_string_fdata qbicDescriptor stringName
```

Parameters

- **qbicDescriptor**–Descriptor returned by qbic_start
- **stringName**–String name for the feature data to be computed.

Example

```
qbic_string_fdata $qb "D100,100:R2,0,80,80,200,100,400"
```

QbDumpDb

Dumps the QBIC dbm files. The full command-line syntax is printed on screen if you invoke the method without any parameters.

Example

```
QbDumpDb -c ibm -d qbic/QbicData -f QbColorHistogramFeatureClass
```

or

```
QbDumpDb -d qbic/QbicData ColorHiF.ibm
```

QbMkDbs

Creates a QBIC catalog. See page 22 for more information. The full command-line syntax is printed on screen if you invoke the method without any parameters.

Example

```
QbMkDbs -c ibm33 -d qbic/QbicData -f QbColorHistogramFeatureClass -f
    QbDrawFeatureClass flower01.jpg flower02.jpg
```

QbMkThmb

Makes thumbnails for QBIC to display the search results. The full command-line syntax is printed on screen if you invoke the method without any parameters.

Example

```
QbMkThmb -p -w 100 -h 100 flower01.jpg flower01.jpg.thumb0.gif
```

QbQBE

Creates a QBIC query. See page 24 for more information. The full command-line syntax is printed on screen if you invoke the method without any parameters.

Example

```
QbQbe -c ibm33 -d qbic/QbicData -f QbColorHistogramFeatureClass -i
    flower01.jpg
```

SoundPlay

This function is available only on Windows NT/95/98. It plays a short wav sound file if the host machine has a sound device.

Syntax

```
SoundPlay waveFileName ?wait?
```

Parameters

- **waveFileName–**Name of a wav file to play.
- **wait**–Optional argument. Instructs qbictcl to wait until the sound file finishes playing before interpreting the next tcl command.

Example

```
SoundPlay QbicSound.wav
```

SoundStop

This function is available only on Windows NT/95/98. It stops playing the wav file initiated by the SoundPlay call. If the sound file has already stopped playing, this call is ignored.

Syntax

```
SoundStop
```

Parameters

None

Example

```
SoundStop
```

start_browser

Starts a browser. On a UNIX platform, the browser should be Netscape and the Netscape program must be in the search path for the user who invokes qbictcl. On NT/95 platforms, the browser will default to Netscape. If Netscape is not installed, it will try Internet Explorer. This function call is not available on Macintosh.

Syntax

```
start_browser ?URL?
```

Parameters

URL–Specify an URL for the Netscape browser

Example

```
start_browser “http://wwwqbic.almaden.ibm.com”
```

Replacing the Mini Web Server

B

Included with the QBIC installation package is a “mini” web server that you can use with the QBIC demo application. You can also use the mini web server when developing and testing your own QBIC-based applications. However, when you deploy your application, it is recommended that you replace the mini web server with a full-scale server.

This appendix describes how to replace the mini web server with a full-scale server.

There are a number of full-scale web servers available. See “General Guidelines for Setting Up a Web Server” on page 181 for general information on how to configure a web server for QBIC. The following sections give specific examples for the following web servers:

- “Guidelines for the IBM Internet Connection Server” on page 182
- “Guidelines for the Apache Web Server” on page 182
- “Guidelines for the Microsoft Peer Web Server” on page 183

If the full-scale web server that you want to use is not described in this appendix, the general guidelines should help you to set up the parameters required to run your QBIC application. See your server’s documentation for details on setting up the server.

NOTE: *The examples in this appendix assume that QBIC is installed in the QBICDIR directory. For example, the QBICDIR directory can be under /usr/local/qbic on a UNIX system and under c:\usr\local\qbic on a PC. The QBIC installation directory path is required to set up the web server.*

QBIC uses an extended tcl script as its CGI script. When you configure your full-scale web server, you need to copy tcl scripts from the `QBICDIR/script` directory to the cgi-bin directory of the full-scale web server.

For each catalog, the corresponding script is called *catalog*.tcl in `QBICDIR/script`, where *catalog* is the catalog name. For example, QBIC comes with a script called `ibm33.tcl`, which corresponds to the `ibm33` catalog.

General Guidelines for Setting Up a Web Server

When setting up a web server, you need to add environment variables and to map directories. In most cases, you make these configuration changes by editing the server’s configuration files. Check your server’s documentation for details.

Add the following environment variables:

```
TCL_LIBRARY QBICDIR/script
QBTCL_LIB QBICDIR/script
```

Map the following directories:

Alias/classes	QBICDIR/classes
Alias/images	QBICDIR/html/images
Alias/gifs	QBICDIR/html/gifs
Alias/thumb	QBICDIR/html/thumb
Alias/cgi-bin	QBICDIR/script

Guidelines for the IBM Internet Connection Server

For the IBM Internet Connection server, edit the `httpd.cnf` file, which is located in `ROOT/httpd.cnf`. `ROOT` is the location where you installed the operating system. For example, for the Windows NT operating system, `ROOT` is `\winnt`, and for the Windows 95 operating system, `ROOT` is `\windows`.

Map the following directories in the `httpd.cnf` file:

pass/classes/*	QBICDIR\classes*
pass/images/*	QBICDIR\html\images*
pass/gifs/*	QBICDIR\html\gifs*
pass/thumb/*	QBICDIR\html\thumb*
pass/cgi-bin/*	QBICDIR\script

In Windows 95, edit the `autoexec.bat` file to set the following environment variables.

In Windows NT, go to **Control Panel→ System→ Environment tab** to set the following environment variables.

```
Set TCL_LIBRARY=QBICDIR\script
Set QBTCL_LIB=QBICDIR\script
```

Guidelines for the Apache Web Server

For the Apache server, edit the httpd.conf file, which is located in `HTTPD/conf/httpd.conf`. `HTTPD` is the location where you installed the server. You will need root privileges to edit this file.

Map the following directories in the `httpd.cnf` file:

`Alias/classes`	`QBICDIR/classes`
`Alias/images`	`QBICDIR/html/images`
`Alias/gifs`	`QBICDIR/html/gifs`
`Alias/thumb`	`QBICDIR/html/thumb`
`Alias/cgi-bin`	`QBICDIR/script`

Add the following environment variables to the `httpd.conf` file:

```
SetEnv TCL_LIBRARY QBICDIR/script
SetEnv QBTCL_LIB QBICDIR/script
```

Guidelines for the Microsoft Peer Web Server

For the Microsoft Peer web server, after performing all of the procedures in this section, reboot the NT system for the changes to take effect.

Set the following environment variables under **Control Panel→ System→Environment tab:**

Environment variable	Value
`QBTCL_LIB`	`QBICDIR\script`
`TCL_LIBRARY`	`QBICDIR\script`

Map the following directories. The steps in this section describe how to map the first directory. Map the others using the same steps.

NOTE: *Make sure you check the execute box for the "cgi-bin" directory mapping.*

`/classes`	`QBICDIR\classes`
`/images`	`QBICDIR\html\images`
`/gifs`	`QBICDIR\html\gifs`
`/thumb`	`QBICDIR\html\thumb`
`/cgi-bin`	`QBICDIR\script`

1. Start the Internet Service Manager by selecting **Start→ Programs→ Microsoft Peer Web Services→ Internet Service Manager**.
2. Double-click **Icons** next to **WWW service**.
3. Choose the **Directories** tab.
4. Click **Add**.
5. In the **Directory** text field, enter `c:\usr\local\qbic\classes`.

6. In the **Virtual** directory text field, enter `/classes`.
7. Do the same for the other directories.

You also need to modify the Windows registry to allow script mapping:

1. Go to `c:\WINNT` (or wherever you installed Windows NT).
2. Type **RegEdit**.
3. From within the Registry Editor, go to **HKEY_LOCAL_MACHINE→ System→ Currentcontrolset→ Services→ W3svc→ Parameters→ Scriptmap**.
4. From the **Edit** menu, select **New→ String Value**.
5. Enter `.tcl` as the **Name** and `QBICDIR\bin\qbictcl.exe %s %s` as the **Value**.

Sample Code

C

There are sample programs in the `qbic/QbicApi` directory that illustrate how various QBIC class methods are implemented. This appendix contains the following sample programs:

- QbMkDbs.cpp for Database Population
- “QbQBE.cpp for Database Query” on page 186
- “QbMkThmb.ccp for Generating Thumbnails” on page 186
- “QbDumpDb.cpp for Dumping a Database” on page 186

These sample applications are used for QBIC database population and query. They are included to demonstrate how easy you can include the QBIC functionality into your application.

If you need to find out how they are implemented, you should go to `qbic/QbicApi` to study `QbicWrap.hpp` and `QbicWrap.cpp`.

QbMkDbs.cpp for Database Population

```
#include "QbicWrap.hpp"

int main(int argc, char **argv)
{
    QbicWrapClass qbicMake;
    int rc;

    rc = qbicMake.QbicWrapDBS(argc, argv);
    if(rc) {
       QbPrintErrorMessage(stderr);
       return -1;
    }
    return 0;
}
```

QbQBE.cpp for Database Query

```
#include "QbicWrap.hpp"

int main(int argc, char **argv)
{
    QbicWrapClass qbicMake;
    int rc;

    qbicMake.SetScreenPrint(1);
    rc = qbicMake.QbicWrapQBE(argc, argv);
    if(rc) {
       QbPrintErrorMessage(stderr);
       return -1;
    }
    return 0;
}
```

QbMkThmb.ccp for Generating Thumbnails

```
#include "QbicWrap.hpp"

int main(int argc, char **argv)
{
    QbicWrapClass qbicMake;
    int rc;

    qbicMake.SetScreenPrint(1);
    rc = qbicMake.QbicWrapThm(argc, argv);
    if(rc) {
       QbPrintErrorMessage(stderr);
       return -1;
    }
    return 0;
}
```

QbDumpDb.cpp for Dumping a Database

```
#include "QbicWrap.hpp"

int main(int argc, char **argv)
{
    QbicWrapClass qbicMake;
    int rc;

    rc = qbicMake.QbicWrapDumpDb(argc, argv, stdout);
    if(rc) {
       QbPrintErrorMessage(stderr);
       return -1;
    }
    return 0;
}
```

Index

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

W

Z