IBM DB2 Information Integrator

Wrapper Developer's Guide

Version 8.2

SC18-9174-00

IBM DB2 Information Integrator

Wrapper Developer's Guide

Version 8.2

SC18-9174-00

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 139.

Contents

About this book

This books helps you develop wrappers to access custom data sources or data sources not covered by the standard set of wrappers that IBM offers.

Who should read this book?

Data administrators, information analysts, system integrators, Web integrators, data librarians, data architects, and application developers who are developing wrappers to access custom data sources for IBM DB2 Information Integrator.

Conventions and terminology used in this book

IBM DB2 Information Integrator uses standard terminology for concepts about database, connectivity, Structured Query Language (SQL), and local area network (LAN). All the DB2 Information Integrator concepts that are used in this book are defined in the glossary. Unless otherwise specified, assume the following meanings:

Data A raw fact. It can be structured, unstructured, or semi-structured. Data is usually organized for analysis. Data also helps you to make decisions.

Information
Data in a usable form, usually processed or interpreted in some way.

Part 1. Overview of federated concepts and developing wrappers

This part of the book provides an overview of the main federated concepts that you need to be familiar with before developing wrappers and provides an overview of the overall wrapper development process.

Chapter 1. Overview of federated concepts

The following topics provide an overview of the main federated concepts that you need to be familiar with before developing wrappers.

Why develop a wrapper?

The problem: No easy way to access or integrate heterogeneous data

Enterprises store data in different brands of relational data management systems (RDBMSs), nonrelational servers, and specialized application systems. This proliferation of servers came about through normal changes in the information technology (IT) industry. These changes include the merging of IT shops, gradual upgrades to products and solutions, and the need for applications to handle specialized data.

Of course, there are benefits to consolidating data from different data sources and thereby providing a new view of this data. For example, you might join customer data in the federated server with information that is stored in a customized geographical information systems (GISs). This would enable you to map where customers live who have specific preferences. Or, perhaps as the result of a merge of divisions of a company, you might need to retrieve data stored in different servers from both divisions.

Unfortunately, one result of data storage heterogeneity is user and application interfaces designed to access a limited subset of servers. So if most of your data is accessible through one interface you will still have some critical data that this interface cannot access. You would then require a second interface to access the critical data. To do this, you might need to write special applications to integrate both sets of data into the same result set.

The solution: Federated systems

DB2® Information Integrator's solution is to enable you to set up a distributed computing system, called a *federated system*. Through a federated system clients can query data sources of all kinds from a single interface. A federated server instance called the *federated server* manages the system. The data sources are sometimes referred to as *foreign servers*.

To DB2 UDB clients (end users and applications), the data sources in a federated system appear as a single collective database. Actually, the clients interface with a DB2 UDB database, called the *federated database*. The federated database is managed by the federated server. To retrieve data from data sources, a client submits a query in DB2 UDB's SQL dialect to the federated database. This query references tables or other data stores within the data sources. It can request results in the form of tables or views that combine output from any number of these data sources.

Example: Accessing and integrating data

Suppose that two nonrelational data sources in a federated system contain life sciences information. In one a table contains data about biochemical experiments and their results. In the other a data set contains information about molecules that are candidates for new drugs. The federated server has its own identifiers for the table and data set—EXP and MOLECULES, respectively. Such identifiers—names by which the federated server refers to tables, data sets, and other kinds of data stores in a foreign server—are called *nicknames*.

Suppose that a user wants to find molecules with a structure similar to the structure of molecules that yielded specific results in a stomach experiment. To do this, the user writes a query that references the table and data set by their nicknames. Figure 1 shows this query.

```
SELECT M.ID, E.MOLECULE_ID, E.RESULTS
FROM MOLECULES M, EXP E
WHERE E.EXP_TYPE = "STOMACH"
AND E.RESULTS > 0.8
AND SIMILAR_TO(E.MOLE_ID, M.ID) > .85
```

Figure 1. Query to request IDs of molecules similar to molecules with a result > 0.8 in a stomach experiment.

The client submits this query to the federated database. The federated server consults its system catalog and finds that EXP is a nickname for a table in the first server. It finds that MOLECULES is a nickname for a data set in the other server. The federated server learns from the catalog that the wrapper designated to retrieve data from the first server is called EXPERIMENTS. It also learns from the catalog that the wrapper designated to retrieve data from the second server is called MYMOL.

The federated server and the wrappers cooperatively develop an execution plan for the query. On the basis of this plan, the federated server decomposes the query into fragments. The wrappers then use the servers' respective application programming interfaces (APIs) to push the fragments down to the servers. The servers return results. The wrappers retrieve the results from the APIs and pass them to the federated server. The federated server then consolidates the results and returns them to the client.

A walk through a basic federated query

At a high level, the basic steps in a federated query are:

1. User or application submits a query.
2. The federated server decomposes the query by source.
3. The federated server and wrappers collaborate on a query plan.
4. The federated server implements the plan through query execution.
5. Wrappers take sources through each source's API.
6. Sources return data to the wrappers.
7. Wrappers return the data to the federated server.
8. The federated server compensates for work that the data sources are unable to do and combines data from different sources.
9. The federated server returns data to the user or application.

After a query is submitted, The federated server consults its system catalog. It is looking for information such as what tables or other data stores contain the information to be retrieved and what wrappers have been designated to initiate the retrieval.

The federated server devises alternative strategies, called *access plans*, for evaluating the query. Such a plan might call for parts of the query to be processed by the data sources, by the federated server, or partly by the sources and partly by the federated server. The federated server chooses among the plans primarily on the basis of cost.

The optimizer generates sub-pieces of the original query submitted by the user's application called *query fragments*. The federated server submits each query fragment to a wrapper in a *request*. The wrapper responds with a *reply*. The reply lets the optimizer know which *sub-fragments* (such as select-list elements and predicates) of that specific query fragment the wrapper can execute. This set of sub-fragments is called the *accepted fragment*. In the reply, the wrapper also gives an estimate of the cost (in time) and number of rows that will be produced if it is asked to evaluate the accepted fragment. The optimizer then *compensates* for those sub-fragments that the data source can not handle by adding them to the federated server's portion of the query plan. Overall, the whole process by which the wrapper and the federated server interact during query planning is called the *Request-Reply-Compensate (RRC)* protocol .

The federated server analyzes combinations of the plans for individual fragments to determine the best overall plan.

Related concepts:

- "Query processing for federated systems" on page 9
- "Wrapper module" on page 5

Wrapper module

| In C++, the wrapper module is a shared library with specific entry points that
| provide access to a class of data sources.

| A Java™ wrapper is a set of Java classes.

| DB2® UDB loads both the C++ and Java wrapper module on demand dynamically.

The wrapper module is what you will be coding using specific classes that are derived from base classes that are supplied with the federated server. It will contain specific building blocks that allow it to act as a translator between your data source and your federated system.

Figure 2 on page 6 illustrates the parts of a wrapper module.

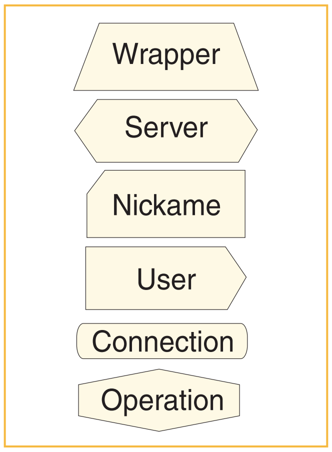

Figure 2. Parts of a wrapper module

Figure 2 shows that the wrapper module contains the following building blocks:

- Wrapper
- Server
- Remote User
- Nickname
- Remote_Connection
- Remote_Operation

| All the building blocks, except the remote connection and remote operation classes,
| represent persistent entities that are registered and described in the federated
| server catalogs.

Table 1. Descriptions, services, and DDL associated with the basic building blocks of a wrapper module

Building block name	Description	Services Provided	Associated DDL
Wrapper	The code module. It is a shared library with specific entry points that represents a class of data sources.	• Bootstrapping and initialization • Access to servers supported by the wrapper.	CREATE WRAPPER
Server	Represents a specific data source supported by the wrapper.	• Access to a set of nicknames • Query planning using the Request-Reply-Compensate protocol.	CREATE SERVER
Nickname	Represents a specific collection of data at a server.	Describes the nickname schema to the federated server at nickname registration or via DDL.	CREATE NICKNAME

Table 1. Descriptions, services, and DDL associated with the basic building blocks of a wrapper module (continued)

Building block name	**Description**	**Services Provided**	**Associated DDL**
User	Provides the information required to authenticate the end user to a particular server.	Maps the federated server user information to source information. For example, user id and password.	CREATE USER MAPPING
Remote_Connection	Represents the federated server's connection to a source Transient: no persistent catalog information.	• Connect and disconnect to/from source • Transaction management.	n/a
Remote_Operation	Represents an active operation at a source (for example query, insert, update, delete). Transient: no persistent catalog information. Multiple operations can be in progress.	• Remote query: read-only query of data • Passthru: direct session with data source (Source's language using source's names).	n/a

Related concepts:

- "Query processing for federated systems" on page 9
- "Wrappers and wrapper modules" in the *Federated Systems Guide*
- "Why develop a wrapper?" on page 3

How users add data sources to federated systems

From the user or application's perspective, adding a data source involves:

- Telling the federated server how the external data will be represented as rows and columns in the relational model
- Configuring the wrapper for the data source so that it can communicate with the source to retrieve its data

The registration process is done through a series of DDL statements your end user or application program issues.

When your user submits a DDL statement to the federated server, the associated wrapper code verifies the correctness of the statement. The wrapper can augment the DDL information with information from the data source or through hardcoded information. The data source can provide the wrapper with information such as configuration parameters and statistics. Parameters can also be called host variables, input variables, or input data. After the wrapper verifies the information from the DDL statement is valid, and possibly augments the information, the federated server stores the information in the appropriate catalogs.

Options on each DDL statement allow you to use a wrapper across several different variations of the same data source.

For example, a wrapper for flat files could have an option that specified the path to the flat file. In that way, one wrapper can deal with flat files in various paths.

The steps that your user or an application has to complete to add a new data source are:

1. Register the wrapper using the CREATE WRAPPER DDL statement.
   a. Determine what wrapper options this wrapper supports.
   b. Determine values for wrapper options.
   c. Issue the CREATE WRAPPER DDL statement

      For example,

      ```
      CREATE WRAPPER Dctm_Wrapper LIBRARY 'libdb2lsdctm.a';
      ```
2. Register a server
   a. Determine what server options this wrapper supports.
   b. Determine values for server options.
   c. Issue the CREATE SERVER DDL statement

      For example,

      ```
      CREATE SERVER Dctm_Server1
      TYPE DCTM
      VERSION 3
      WRAPPER Dctm_Wrapper
      OPTIONS(NODE 'Dctm_Docbase',
      OS_TYPE 'AIX',
      RDBMS_TYPE 'ORACLE');
      ```
3. If needed, define remote users for that server using the CREATE USER MAPPING DDL statement

   For example,

   ```
   CREATE USER MAPPING FOR Chuck SERVER Dctm_Server1
   OPTIONS(REMOTE_AUTHID 'Charles',REMOTE_PASSWORD 'Charles_pw');
   ```
4. Register any specialized functions of that server:
   a. Create a local template function using the CREATE FUNCTION... AS TEMPLATE DDL statement

      For example,

      ```
      CREATE FUNCTION DCTM.ANY_EQ (CHAR(),CHAR())RETURNS INTEGER
      AS TEMPLATE DETERMINISTIC NO EXTERNAL ACTION
      ```
5. Create nicknames for data sets
   a. Determine what column names and column types you will specify on the CREATE NICKNAME statement.
   b. Determine what nickname options and column options this wrapper supports.
   c. Determine values for the nickname options and column options.
   d. Issue the CREATE NICKNAME DDL statement

      For example,

      ```
      CREATE NICKNAME std_doc (
      object_name varchar(255)not null,
      object_id char(16)not null OPTIONS(REMOTE_NAME 'r_object_id'),
      object_type varchar(32)not null OPTIONS(REMOTE_NAME 'r_object_type'),
      ```

```
       title varchar(255)not null,
       subject varchar(128)not null,
       author varchar(32)OPTIONS(REMOTE_NAME 'authors',IS_REPEATING 'Y'),
       keyword varchar(32)OPTIONS(REMOTE_NAME 'keywords',IS_REPEATING 'Y'),
       creation_date timestamp OPTIONS(REMOTE_NAME 'r_creation_date'),
       modifed_date timestamp OPTIONS(REMOTE_NAME 'r_modify_date'),
       status varchar(16)not null OPTIONS(REMOTE_NAME 'a_status'),
       content_type varchar(32)not null OPTIONS(REMOTE_NAME 'a_content_type'),
       content_size integer not null OPTIONS(REMOTE_NAME 'r_content_size'),
       owner_name varchar(32))
       FOR SERVER Dctm_Server2 OPTIONS (REMOTE_OBJECT 'dm_document',
                   IS_REG_TABLE 'N')
```

6. Query data by using standard SQL statements.

   For example,

   ```
   SELECT object_name
   FROM std_doc
   WHERE DCTM.ANY_EQ(author,'Joe Doe')=1
   ```

When writing the wrapper code, you are responsible for defining what these DDL statements look like within a set of constraints. You are responsible for determining what existing options to use and creating any new ones that are needed for your data source.

Related concepts:
- "Why develop a wrapper?" on page 3

Related tasks:
- "Adding data sources to the DB2 Control Center" on page 119

Query processing for federated systems

There are two phases of query processing: query planning and query execution. The query planning phase occurs during compile-time. The query execution phase occurs during runtime. Figure 3 on page 10 shows the flow of query processing in a federated system during compile-time and runtime.

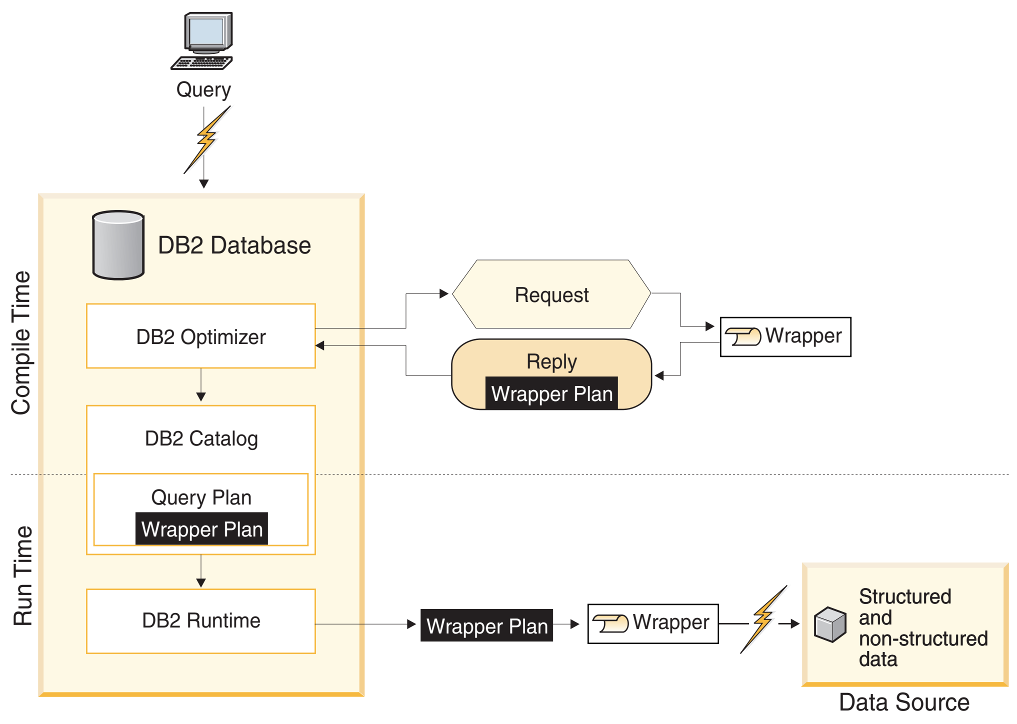

Figure 3. Federated query processing flow

Related concepts:

- "Query execution for federated systems" on page 16
- "Wrapper module" on page 5

Request-reply-compensate protocol

During query planning, the optimizer generates sub-pieces of the original query submitted by the user's application called *query fragments*. A query fragment can contain tables, predicates, and head expressions. Head expressions are expressions found in the SELECT clause of a query. The optimizer then submits each query fragment to a wrapper in a *request*.

By definition, all the data needed to evaluate a fragment comes from one data source. However, the processing of this data can be done by the foreign server, by the federated server, or some by each. The wrapper indicates which sub-pieces of the fragment, called *sub-fragments*, it can evaluate, and puts this information in the *reply* to the request. The wrapper must either accept or reject an entire sub-fragment.

The reply contains:

- accepted nicknames, predicates and head expressions
- cost and cardinality estimates for the accepted fragment
- ordering properties, if any, for the results that will be returned
- Wrapper Execution Descriptor (described in the following text)

For the purposes of this document, "quantifier" and "nickname" are interchangeable.

For a single query, the optimizer typically generates many requests for each wrapper, each request representing a different fragment of the original query. For each such request, the wrapper generates zero, one, or more replies. Each reply represents a different accepted fragment. An accepted fragment is a fragment the wrapper or data source can evaluate itself. Each reply contains the associated cost and cardinality estimates for the accepted fragment.

By the end of query planning, the optimizer will have made a cost-based decision. It will have come up with a *plan* incorporating some set of the accepted fragments offered up by the wrapper in response to requests. It is this particular set of accepted fragments that the wrapper will eventually be asked to execute.

The optimizer will ensure that any sub-fragments that are not part of the accepted fragments in the plan it has chosen will be evaluated by the federated server. Examples of this include a complex predicate or a sort that is beyond the capability of the data source in question. This includes all cross-source joins as well as any other expressions (such as function invocations) that mix data from multiple sources since a fragment is single-source. This is called *compensation*.

Overall, the whole process by which the wrapper and the federated server interact during compilation is called the *Request-Reply-Compensate (RRC)* protocol.

The wrapper can determine the cost and cardinality estimates to put in the replies on its own or it can reuse a default cost model provided by the federated server. It can also selectively replace parts of the default cost model so as to improve its accuracy without building a new model from scratch. The default model uses statistics that the wrapper can calculate from the data, or they can be supplied via DDL.

As part of the reply, wrappers also have to provide a *Wrapper Execution Descriptor*. This is a black box whose content is up to the wrapper. The wrapper must be able to submit the accepted fragment that the reply represents to the data source. If the optimizer uses the accepted fragment in the execution plan it ultimately selects, the Wrapper Execution Descriptor will be returned to the wrapper when it is time to run the query. In between, the Wrapper Execution Descriptor resides in the federated server catalogs, as part of the access plan for a precompiled query.

The wrapper for the source has complete control over the persistent representation of the remote query in the Wrapper Execution Descriptor.

Manipulating requests and replies with handles

DB2 UDB provides a functional interface to manipulate requests and replies.

A request corresponds to an SQL query, but the query is not represented in SQL because the data sources do not understand SQL. Instead, integer handles are used to identify and navigate query expressions. Predicates and head expressions are represented as operator trees. Functions are provided to describe each kind of node and navigate to any of the node’s children. Methods are provided to move terms from request to reply.

Example of Request-Reply-Compensate protocol

For example, consider the example in Figure 4 on page 12.

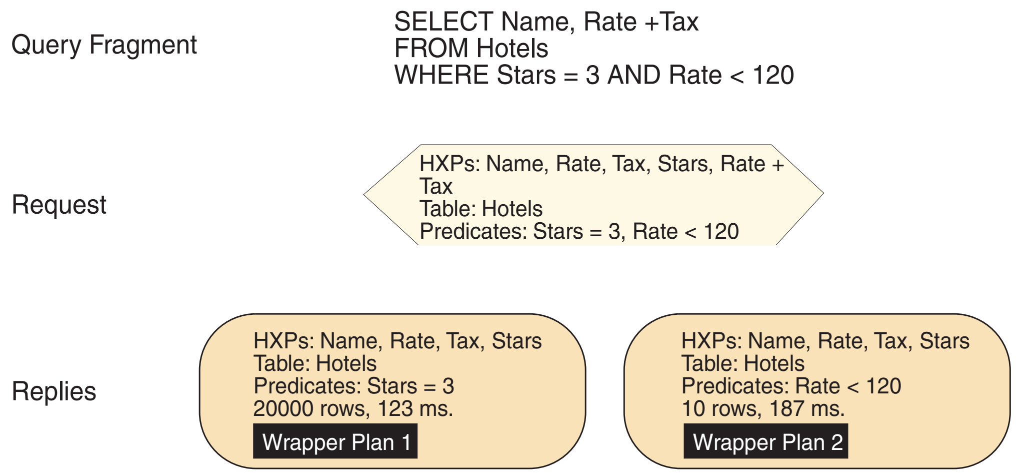

Figure 4. Request-Reply-Compensate protocol example

The figure shows the RRC protocol in action. In this scenario, you want to access a data source that is a Web site that only allows you to specify one predicate in a request to the web server.

The figure shows a query fragment of a single-table access query. Note that the SELECT list contains an expression: rate + tax.

In the request, note that the federated server can request the columns contained in predicates and expressions individually in the list of head expressions: rate, tax, stars.

There are two replies with different predicates accepted, costs and cardinalities, and wrapper plans (aka wrapper execution descriptors). Neither plan accepts the head expression, rate + tax. Since the wrapper and data source are not able to handle this head expression, the federated server will compute its value from the individual column values for rate and tax.

Related concepts:
- "Query processing for federated systems" on page 9
- "Query execution for federated systems" on page 16
- "Control flow for query execution" on page 57

Default cost model for federated queries

This section describes how wrappers use a cost model to provide costing information to the federated server optimizer.

Query costs and query planning

Through the Reply class, the wrapper provides information to the federated server. The federated server uses this information when determining the most appropriate plan to process a particular query. The Reply class provides methods that calculate the following four pieces of information:

1. The cardinality of the query fragment represented by the reply. This is an estimation of the number of rows that the query fragment will return.

2. An estimate of the time, in milliseconds, to retrieve the first tuple selected by the query fragment. This estimate is for the first invocation of the query fragment. It is referred to as the fragment's *first-tuple cost*.
3. An estimate of the time, in milliseconds, to retrieve an entire answer set selected by the query fragment. This estimate is for the first invocation of the query fragment.
4. An estimate of the time, in milliseconds, to retrieve an entire answer set selected by the query fragment. This estimate is for the second or subsequent invocation of the query fragment, possibly with new parameters bound in. It is referred to as the fragment's *re-execution cost*. The re-execution cost will differ from the first-tuple cost if there is preprocessing a wrapper needs to do the first time it submits a query fragment to a remote source. If the wrapper submits the same fragment, it will not need to repeat the preprocessing.

A wrapper can sub-class the Reply class and provide its own mechanism for calculating these costs, or it can use the default cost model the Reply class provides. The following section describes the default cost model.

The default cost model

The cost model the default Reply class implements calculates the four pieces of information that were listed previously using a default set of cost equations driven by statistical information for each nickname involved in the query fragment. A wrapper using the default cost model must provide four statistics, listed in the following section, for each nickname.

Statistics for the default cost model

By default, the federated server stores these statistics in the system catalog. By overriding the appropriate methods, a wrapper can provide these in a different manner. The wrapper can override these methods while retaining the rest of the default cost model.

The four statistics are:

1. The cardinality of a nickname. This is defined as the number of rows contained in the nickname. The federated server stores the cardinality for an individual nickname in the system table SYSCAT.TABLES or SYSSTAT.TABLES (the ″CARD″ column in either table.) If cardinality is not available for a nickname, the cost model uses a default value of 1000 rows.
2. The setup cost for a nickname. Setup cost represents the typical time, in milliseconds, that it takes a wrapper to get a query fragment ready to submit to the remote source. Setup begins when a wrapper receives the wrapper Execution Descriptor it produced during query planning, and ends when the wrapper is ready to submit the corresponding operation to the remote source. Setup cost should only include work that the wrapper does not need to repeat if the wrapper is asked to perform the same query fragment again, perhaps with a different parameter value. For example, if a wrapper submits query fragment to a remote source in the form of a URL, setup cost includes the time required to generate that URL from the information stored by the wrapper in the Execution Descriptor. The federated server stores this statistic in the SETUP_COST nickname option. If that option is not present for a nickname, then the cost model uses a value of 25 milliseconds.
3. The submission cost for a nickname. Submission cost represents the typical time, in milliseconds, that it takes a wrapper to submit a query fragment to the remote source. Submission begins at the end of setup, as defined above, and

ends when the wrapper is ready to request the first row or block of result data from the source. Submission cost should only include work that the wrapper must repeat each time a given query fragment is submitted. For example, if a new HTTP connection is required for each interaction with the remote source, submission cost should include the time necessary to create this connection. The federated server stores this statistic in the SUBMISSION_COST nickname option. If that option is not present for a nickname, then the cost model uses a value of 2000 milliseconds.

4. The advance cost for a nickname. This is the typical time, in milliseconds, that it takes to fetch a single row for the nickname. It is exclusive of any time necessary to start a query. The federated server stores this statistic in the ADVANCE_COST nickname option. If that option is not present for a nickname, then the cost model uses a value of 50 milliseconds. If the data source returns data in blocks, rather than rows, calculate the advance cost by dividing the typical cost of fetching a block by the typical number of rows per block.

Although you could obtain the required statistics by instrumenting your wrapper code, it is usually easier to obtain them by external measurement of the running time of typical queries against the nickname in question. For example, you can find the advance cost by running two queries that differ in the number of results each query returns, and dividing the difference in execution time by the difference in the number of rows each query returns. This applies especially to the three cost statistics. Frequently, data sources have direct methods of determining cardinality.

Default cost equations

The default cost model provides a set of four cost equations that derive the four parameters the optimizer requires. The following list describes each equation.

Cardinality

The cardinality for a query fragment is calculated in two steps. First, the cost model obtains the cardinality for each of the nicknames and multiplies these values together.

In the second step, the cost model multiplies the total number of rows that are calculated in the first step by the selectivity of the predicates in the query fragment. The selectivity is a number between 0.0 and 1.0 that reflects the degree to which the predicates filter rows from the nicknames out of the result set, with a small value indicating a greater degree of filtering. The default cost model provides a method that uses algorithms supplied by the federated server to estimate predicate selectivity. A wrapper can override this method if it is able to supply a more accurate selectivity estimate. A key contributor to errors in selectivity is correlation among predicates in a set, of which the default model is completely unaware.

For example, although the selectivity of the predicate `Make='Carmaker1_make'` is 0.13, and the selectivity of `Model='Carmaker1_model'` is 0.05, the combined selectivity of `Make='Car1_make' AND Model='Car1_model'` is also 0.05, because the values of the attributes `Make` and `Model` are not independent: every car model is from the same car maker. If your wrapper is aware of correlations between attributes, you could override the default selectivity estimation method. You could also provide your own selectivity estimation method when the distribution of values for an attribute is highly skewed. This can be done while keeping the rest of the default cost model. Furthermore, note that if you do provide a custom selectivity estimation method, the federated server optimizer will also call your custom selectivity estimation method to calculate the selectivity of any predicates that the wrapper did not accept in the reply. As stated previously, the wrapper could be aware of skew or correlation among predicates, and can therefore provide a better estimate, even when the predicates cannot be evaluated by the source as part of the accepted query fragment.

First tuple cost
The first tuple cost is the sum of three values. The first is the average of the setup costs for all nicknames in the query fragment. The second is the average of the submission costs for all the nicknames in the query fragment. The third is the average of the advance cost for all nicknames in the query fragment.

Total cost
The total (first answer set) cost is the sum of three values. The first is the average of the setup costs for all nicknames in the query fragment. The second is the average of the submission costs. The third is the product of the average of the advance costs multiplied by the estimated cardinality of the query fragment.

Re-execute cost
The re-execute cost for a query fragment is the sum of two values. The first is the average of the submission costs for all nicknames in the query fragment. The second is the product of the average of the advance costs multiplied by the estimated cardinality of the query fragment. Note that in calculating the execution-time estimates, the cost equations typically use the average value of a statistic across all the nicknames in the query fragment. If this is unlikely to give reasonable results for your data source, consider overriding the cost equations by subclassing the Reply object.

Costing and query planning summary

You have several options when it comes to calculating costs for query fragments. Figure 5 on page 16 illustrates these options.

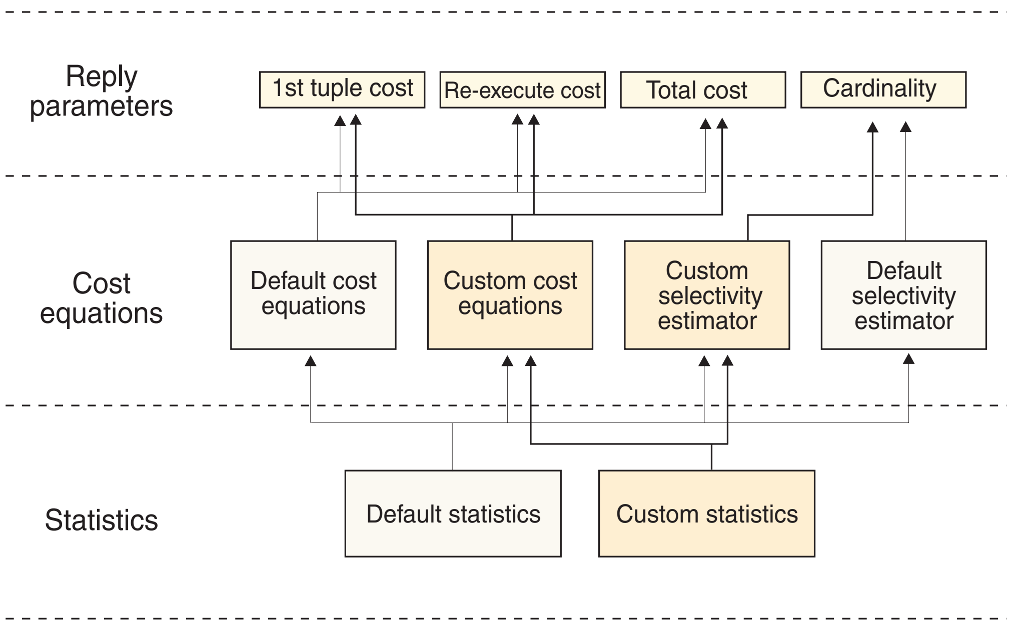

Figure 5. How costing works

In approximate order of increasing complexity and increasing power and flexibility, these are:

1. Accept the default cost model, as is; or
2. Override how the four statistics (cardinality, average setup cost, average submission cost and average advance cost) for a nickname are calculated; or
3. Override the default selectivity calculation; or
4. Replace the entire cost model by subclassing the Reply object.

Related concepts:

- "Relative cost of queries for the data source" on page 26
- "Control flow for query planning" on page 55

Query execution for federated systems

Once query planning is complete, the federated server can run the query. First, the federated server distributes the query fragment assigned to each data source to the corresponding wrappers. In turn, the wrappers submit the fragments to the data sources and retrieve their results. The federated server combines and further processes the results. These steps are typical, and can vary depending on how a specific source handles the concept of a connection, whether the source accepts requests via a query language or through some other API, what kind of result set cursors or iterators (if any) are supported by a source, and so forth. Figure 6 on page 17 walks you through the actions that are normally taken to distribute a query, execute it, and retrieve its results.

Typical process of distributing a query, executing it, and returning its results:

1. The federated server passes to the wrapper the authorization information that was specified in the CREATE USER MAPPING statement.
2. The federated server requests the wrapper to establish a connection to the data source.
3. The wrapper establishes the connections that the federated server requested.
4. The wrapper gets the Wrapper Execution Descriptor produced during query planning for this fragment of the query. The Wrapper Execution Descriptor must contain sufficient information to submit the query to the data source.
5. The wrapper submits the translated query fragments to the data sources that the queries reference. The wrapper then obtains an iterator for the result set that the wrapper is to retrieve.
6. The federated server requests a row of results from the wrapper. The wrapper, in effect, "forwards" the request to the appropriate data source.
7. The data source executes a portion of a query fragment in order to return the requested row.
8. The wrapper retrieves the requested row. The wrapper also converts the types of the data in the row to the federated server data types, and copies the converted types into buffers.
9. The federated server, the data source, and the wrapper repeat steps 7, 8, and 9 for each successive row of results.
10. The wrapper retrieves the last row of results. After a number of units of work go by without a reference, the federated server disconnects from the data source.
11. The wrapper disconnects from the data source.

Figure 6. Typical process of distributing a query, executing it, and returning its results

Related concepts:

- "Query processing for federated systems" on page 9
- "Wrapper module" on page 5

Using passthrough with wrappers

Passthrough allows an application to submit a query or other request directly to an external data source. The query or other request uses the data sources native query language. Data can be retrieved as rows and columns using passthrough. Results obtained with passthrough cannot be joined or otherwise combined with results from other sources. Implementation of passthrough is optional. It is useful for debugging connectivity problems, and provides a way for applications to perform administrative commands against an external source.

Related tasks:

- "Remote passthru class" on page 97

Chapter 2. Overview of developing wrappers

The following topics provide an overview of the wrapper development process and the wrapper development kit that you use to develop wrappers.

Wrapper development process

As the writer of the wrapper for a particular data source, you need to know the basic flow of development to create your wrapper module.

Table 2. The phases in the wrapper writing process

Phase	Subtasks
Concepts	Understand federated concepts
	Understand the general wrapper writing process
Design	Understand your data source
	Develop a relational model for your data source
	Decide on options for each federated construct
	Map federated constructs to your data source
	Decide what kinds of queries your data source supports
	Design a cost model
	Design for passthrough
	Design for error handling
Code	Code the wrapper subclasses
	Code for registration
	Code for initialization
	Code for query planning
	Code for query execution
	Code for passthrough
	Consider portability issues
Document	Document your wrapper
Build and package	Compile your wrapper
	Link your wrapper
	Package your wrapper
	Install your wrapper
Test	Test your SQL statements
	Debug and trace your wrapper
	Test your wrapper options
	Test a wide range of queries

Related concepts:

- "Typical procedure for developing a wrapper" on page 61

- "Tips for developing wrappers" on page 63
- "Wrapper development kit" on page 20

Wrapper development kit

DB2® Information Integrator includes a software development kit (SDK) for developing wrappers in C++ and Java™.

The wrapper development kit contains:
- Sample C++ wrapper
- Sample Java wrapper
- Tools and samples for adding wrappers to the DB2 Control Center

The default Windows® directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.

Sample C++ wrapper

Table 3 shows which directory for each platform where the sample C++ wrapper is located.

Table 3. Directory for sample C++ wrapper by platform

Platform	Wrapper installation directory
AIX®	/usr/opt/db2_08_01/samples/wrapper_sdk
HP/Sun/Linux	/opt/IBM/db2/V8.1/samples/wrapper_sdk
Windows	%DB2PATH%\samples\wrapper_sdk

The sample C++ wrapper contains:
- Header files showing the wrapper APIs (wrapper class declarations)
- A file that allows a wrapper to be linked with the federated server
- The wrapper common library (a stub library provided that loads and invokes the custom wrapper's libraries)
- Sample wrapper source code used to demonstrate the use of the C++ API for developing wrappers
- A sample makefile to build the sample wrapper

Sample Java wrapper

Table 4 shows which directory for each platform where the sample Java wrapper is located.

Table 4. Directory for sample Java wrapper by platform

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/samples/wrapper_sdk_java
HP/Sun/Linux	/opt/IBM/db2/V8.1/samples/wrapper_sdk_java
Windows	%DB2PATH%\samples\wrapper_sdk_java

The sample Java wrapper contains:
- Javadoc describing the Java API classes and methods

- Sample wrapper source code used to demonstrate the use of the Java API for developing wrappers

Tools and samples for adding wrappers to the DB2 Control Center

The wrapper development kit includes tools and sample files to help you add support for custom wrappers to the DB2 Control Center:

- The Develop XML Configuration File wizard, which creates a configuration file for adding a custom wrapper to the options in the DB2 Control Center. Table 5 shows which directory contains the file that starts the wizard for each platform.

Table 5. Directory for starting the Develop XML Configuration File wizard by platform

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/lib/db2wrapperconfig
HP/Sun/Linux	/opt/IBM/db2/V8.1/lib/db2wrapperconfig
Windows	%DB2PATH%\bin\db2wrapperconfig.bat

- Sample output files from the Develop XML Configuration File wizard. Table 6 shows which directory contains the sample output files for each platform.

Table 6. Directory for sample output files from the Develop XML Configuration File wizard by platform

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/samples/wrapper_sdk/cc_plugin
HP/Sun/Linux	/opt/IBM/db2/V8.1/samples/wrapper_sdk/cc_plugin
Windows	%DB2PATH%\samples\wrapper_sdk\cc_plugin

- A basic discovery tool, which you can use if you want the wrapper to support the DB2 Control Center's discovery feature. The tool is a simple Java GUI that displays whatever has been discovered for the wrapper's data source. This tool is also included with the DB2 Control Center. Table 7 shows which directory provides the tool as a Java .jar file for each platform.

Table 7. Directory for basic discovery tool by platform

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/tools/db2WrapperDiscoverySDK.jar
HP/Sun/Linux	/opt/IBM/db2/V8.1/tools/db2WrapperDiscoverySDK.jar
Windows	%DB2PATH%\tools\db2WrapperDiscoverySDK.jar

- The sample Java stored procedure provided here is an example of how the build-in discovery can help the wrapper writer to develop the plug-in to the Control Center. Table 8 shows which directory contains the stored procedure, a makefile to compile the stored procedure, and a script to install the markup file into the federated server.

Table 8. Directory for sample Java stored procedure by platform

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/samples/wrapper_sdk\cc_plugin
HP/Sun/Linux	/opt/IBM/db2/V8.1/samples/wrapper_sdk\cc_plugin
Windows	%DB2PATH%\samples\wrapper_sdk\cc_plugin

Related concepts:

- "Wrapper development process" on page 19
- "Typical procedure for developing a wrapper" on page 61

Related tasks:

- "Adding data sources to the DB2 Control Center" on page 119
- "Installing the wrapper development kit" in the *IBM DB2 Information Integrator Installation Guide for Linux, UNIX, and Windows*

Part 2. Designing wrappers for data sources

This part of the book takes you through the following tasks required to design a wrapper:

- Designing a wrapper to work effectively with the data source by determining data source characteristics
- Mapping data sources to federated constructs
- Determining the SQL constructs that the source can accept
- Designing error handling for wrappers

Chapter 3. Determining data source characteristics

Before you begin designing your wrapper, you need to understand your data source. You need to answer the following questions:

- Does the data source offer a choice of APIs?
- What kinds of operations does each interface support?
- Does the data source contain metadata that describes the schema or other properties of the primary data?
- What kinds of queries can the data source answer?
- Can some of the queries be answered more efficiently than others?
- If there can be more than one instance of the data source, what varies from instance to instance and what is always the same?
- Does the data source support client-server communication?
- Is there a data source client available for the platform on which your federated server is to run?
- Does the data source support a transaction model? If so, does it support distributed (one-phase) commit protocols?
- How does the data source authenticate users?
- Does your data source support large objects?

The following sections provide more details and examples for each of the previous questions.

Selection of APIs for the data source

When you study a data source's API, look for:

- Capabilities that the API supports. For example, can you submit the kind of queries you need to submit through this API? Does the API give the wrapper access to metadata describing the schema or statistical properties of data that are managed by the data source? Does the API support transactions? Even though access to nonrelational sources is read-only, a wrapper can participate in a transaction in which relational sources are updated.
- Information that the API requires. For example, what information does the API require when the wrapper is ready to establish a connection to the data source? What information does it require to facilitate retrieval of result sets?

A data source client that supports the selected API must be available on each platform on which you want to run a DB2 UDB federated database server that will access the data source.

Related concepts:

- "Operations that are supported by the interface of the data source" on page 25
- "Metadata at the data source" on page 26

Operations that are supported by the interface of the data source

Use the following questions to help you understand the operations supported by each interface of your data source:

- Does the data source allow you to selectively retrieve information that is based on data values? Can the data source apply predicates?
- Does the data source allow you to retrieve partial entities? Does the data source support projection?
- Can the data source combine data from multiple collections that are based on links or matching data values? Can it do joins?
- Does the data source support any special search capabilities that are not expressible in standard SQL? What mapped functions will be necessary to model the source's search capability?
- Does the data source just return stored data values, or can it return the values of expressions that combine these values with other values or constants?
- Do the data source semantics for predicates match the federated server semantics?

Related concepts:
- "Metadata at the data source" on page 26
- "Relative cost of queries for the data source" on page 26
- "Deciding on nickname and column options" on page 29

Related tasks:
- "Selection of APIs for the data source" on page 25

Metadata at the data source

The more information you can obtain directly from the data source, the less work needs to done by the database administrator when registering the source. In particular, details of the schema can be omitted from the CREATE NICKNAME statement if they can be obtained from the source. Similarly, you can sometimes obtain from the data source statistical metadata used by the wrapper's cost model (custom or default).

Related concepts:
- "Operations that are supported by the interface of the data source" on page 25
- "Multiple instances of the data source" on page 27
- "Client-server communication for the data source" on page 27

Relative cost of queries for the data source

It is important to understand the relative costs of different queries that your data source can answer. Given a query fragment from the federated server, there are often several ways to execute all or part of this fragment at the data source. The wrapper must be able to provide accurate cost estimates for each alternative.

Related concepts:
- "Operations that are supported by the interface of the data source" on page 25

Related tasks:
- "Selection of APIs for the data source" on page 25

Multiple instances of the data source

Most data sources are not truly one of a kind. If they were, all the information required to connect to and use a particular source could be hard coded in the wrapper, including the schema. In practice, one must allow the DBA to customize the wrapper for particular instances of your data source. Since this information will largely be specified via DDL, an important part of wrapper design is to determine what information varies from instance to instance. You can use this information to define a set of options that will allow the DBA to specify what is needed. If the schema varies from instance to instance, it could be possible to hard-code parts of it and specify the rest via DDL.

Related concepts:

- "Metadata at the data source" on page 26
- "Client-server communication for the data source" on page 27

Client-server communication for the data source

Your wrapper will run on the same computer as the federated server that will incorporate your data source into the federated system. If the data source you wish to access runs on a different computer than the federated server, you must use the data source's client/server communication facilities. The wrapper and the entire federated server are essentially a client of the data source. If the data source does not support client/server communication, you will have to provide this capability as part of, or in addition to, the wrapper itself. Sometimes you can solve the problem by running the data source and the federated server on the same computer.

Related concepts:

- "Transaction models and distributed commit protocol for the data source" on page 27
- "User authentication from the data source" on page 28
- "Large object support from the data source" on page 28

Transaction models and distributed commit protocol for the data source

Transactions are a mechanism for making a group of updates to a database atomic: either the entire group of updates appears to take place simultaneously, or none of the updates take place at all. When a transaction involves multiple data sources, the sources must cooperate to ensure atomicity of the transaction across all the sources. DB2® Information Integrator uses the XA distributed commit protocol to coordinate transaction management among sources. Although the current version of DB2 Information Integrator does not support updates to nonrelational data sources, if your data source supports transactions, your wrapper must participate in transaction management. It must participate in transaction management so that locks obtained at your data source can be dropped when the transaction that locked them completes (whether successfully or unsuccessfully).

Related concepts:

- "Client-server communication for the data source" on page 27
- "User authentication from the data source" on page 28

- “Large object support from the data source” on page 28

User authentication from the data source

Clients of the federated system will authenticate to the federated server, using any of the supported authentication schemes. When the federated server attempts to access your data source on their behalf, it will have to present credentials that your source recognizes and accepts as valid. The federated server stores these credentials as a User Mapping in the federated server catalogs. When designing your wrapper, you must understand what credentials your data source expects, for example userid and password. With this understanding, you can define user mapping options that allow these credentials to be stored in the federated server catalog.

Related concepts:

- “Client-server communication for the data source” on page 27
- “Transaction models and distributed commit protocol for the data source” on page 27
- “Large object support from the data source” on page 28

Large object support from the data source

A large object (LOB) data type is defined as a sequence of bytes with a size that ranges from 0 bytes to about 2 gigabytes. (LOB data types include CLOB, BLOB, and DBCLOB data types.) If the data source supports LOBs, then you’ll need to design and implement your wrapper to support them.

Related concepts:

- “Client-server communication for the data source” on page 27
- “Transaction models and distributed commit protocol for the data source” on page 27
- “User authentication from the data source” on page 28

Related reference:

- “Remote query class” on page 92
- “RemoteQuery class (Java)” in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- “Remote_Query class (C++)” in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Chapter 4. Mapping data sources to federated constructs

The following sections describe how to map the data source to a relational model.

For each of the federated concepts, you need to decide on options that work for your data source.

These options are:
- Generic (attribute, value) pairs for storing wrapper-specific information in the federated server's system catalogs
- Specified for any persistent building block (Wrappers, Servers, Nicknames, Users).

  Within a nickname, each column can also have options.
- Defined and introduced by wrapper writers
- In most cases, uninterpreted by the federated server

Each wrapper defines its own set of options.

Designing for nicknames

No matter what data model your source uses to present its data, you must map it to a relational model. This relational model is then provided to the federated server by a user, a database administrator, or an application through the CREATE NICKNAME DDL statement specific to your data source.

Deciding on nickname and column options

The following example illustrates how to define nickname and column options.

Consider a wrapper for a file server. Before the wrapper submits a query fragment to the server, it determines the path and name of the file whose nickname is referenced in the fragment. So that it can do this, you include an option for the path and name in the same statement that defines the nickname.

For example, suppose you have a file named EMPINFO that is located in the directory structure, root/user/. On your CREATE NICKNAME statement you might create a new option called PATH. Your user would assign the path `root/user/EMPINFO` to the PATH option.

Related concepts:
- "Mapping queryable collections of source data to nicknames" on page 29
- "Mapping hierarchical data structures to nicknames" on page 30

Mapping queryable collections of source data to nicknames

A user registers a collection of foreign server data (for example, a table or view) by running the CREATE NICKNAME statement. The statement includes a definition of the collection that has the same characteristics as the definition of a federated server table. When you register a collection of foreign server data, the federated server defines it in the schema as a federated server table. The nickname specified in the statement serves as the table's name. Before other parts of the definition, such as column names and data types, can be specified, you must determine what attributes of the collection correspond to them. Which attributes correspond to columns? Which federated server data type corresponds most closely to the type of the data source attribute? When you answer such questions, you can define the attributes as their counterparts in a federated server table.

Suppose that an Oracle database contains a table named EMPLOYEES. Before you can register this table, you need to determine what parts of it correspond to the federated server's table columns and what parts correspond to these columns' data types. An easy task because, as both Oracle and the federated server follow the relational data model, it's clear that the Oracle table's columns correspond to the federated server table columns and that the Oracle columns' data types correspond to the federated server data types.

But what if your data source is a file server? Then the question is: How do the files in the server correspond to the federated server tables? One possible approach is to regard each file as a counterpart to a table, each file record as a counterpart to a table row, and each field of a record as a counterpart to a column.

Related concepts:
- "Mapping hierarchical data structures to nicknames" on page 30
- "Deciding on nickname and column options" on page 29

Mapping hierarchical data structures to nicknames

Many data sources contain hierarchically organized collections of data. For example, a set of Projects is contained within the entity representing a Department, which can in turn be one of a set of Departments for a Division, and so on. However, the relational model is flat: you cannot have a column of a table which is itself a nested table. Instead, you must define separate tables for each kind of entity and join these tables to navigate the hierarchical relationships among them. For example, to find the Division to which a Department belongs, or to find all the Projects for a Department. When incorporating sources featuring hierarchically structured data into a DB2® UDB federated system, model each of its hierarchical collections as a set of related Nicknames that can be joined together.

Related concepts:
- "Mapping queryable collections of source data to nicknames" on page 29
- "Deciding on nickname and column options" on page 29

Mapping data types from data sources to DB2 Universal Database

Columns of a Nickname must have basic SQL types, like INTEGER, VARCHAR(), DATE, and so forth.

DB2® Information Integrator does not support user-defined types, distinct types, and reference types as columns of nicknames. You must define columns containing such values using their underlying representational types instead. However, you can define views over the data the nickname represents that make use of these constructs.

For most numeric types that are supported by external sources, the mapping to the corresponding SQL type is straightforward.

However, keep attributes like precision, scale, and range of allowable values in mind when deciding on the best SQL type to represent a numeric value. Conversion of the data source's representation of a value into DB2 UDB's representation of that value, and vice-versa, is the responsibility of your wrapper. The more different types you use, the more conversion code you could have to write.

For data source attributes with character data types, the wrapper writer must choose whether to represent the attribute using a fixed- or variable-length SQL type, and determine an appropriate upper limit on length. Other considerations for character data types include character encoding and codepage issues. Generally speaking, it will be the wrapper's responsibility to convert character data to/from the designated codepage of the federated database.

For managing temporal data, map data types representing temporal values to one of the SQL types : TIME, DATE, or TIMESTAMP.

DB2 UDB provides operators to cast or convert between many pairs of SQL types. Consider simplifying your wrapper by representing data source attributes with a basic type like VARCHAR, and defining views that cast or convert this value to a n appropriate type with more semantics, like FLOAT or DATE.

Related concepts:

- "Mapping queryable collections of source data to nicknames" on page 29
- "Mapping hierarchical data structures to nicknames" on page 30

Related tasks:

- "Mapping parts of a wrapper to classes" on page 67

Modeling data source capabilities with function templates

Your data source can have capabilities that are not in DB2® UDB. In that case you will want to extend SQL to model special capabilities of your data source.

For example, suppose that your data source is a Geographic Information System that can determine the area of a region. SQL does not have an "Area" operator, so you model this operator as a function, and write queries like this:

```
SELECT id, name
FROM Regions
WHERE Area(id) > 200
```

Functions that are executed by a remote data source are called *custom functions*. Mapped functions are used in queries in exactly the same way as User-Defined Functions (UDFs). A mapped function differs from a UDF in that DB2 UDB executes UDFs, whereas an external data source executes custom functions.

Defining a custom function is a one-step process.

1. Create a function template via DDL: CREATE FUNCTION AREA(VARCHAR()) RETURNS INTEGER AS TEMPLATE DETERMINISTIC NO EXTERNAL ACTION
   - In order to distinguish your custom functions from other custom functions, UDFs, or DB2 UDB built-in functions with the same name, consider defining a separate schema for your mapped functions.

Related concepts:
- "Modeling data source capabilities using pseudo columns" on page 32

Related tasks:
- "Default cost model for federated queries" on page 12

Modeling data source capabilities using pseudo columns

Another technique for exposing data source capabilities through a relational interface is through the use of pseudo columns. These are columns that are defined as part of a nickname which can be used to control some aspect of a query. They are usually defined by the wrapper, rather than the end user.

An example would be to create a `wildcard` pseudo column that controlled the wildcard character for text search patterns.

Another example is the `MismatchPenalty` pseudo column of the nicknames for the Blast wrapper. This pseudo column controls one of the Blast search parameters, but itself produces no value. A predicate of the form `MismatchPenalty = 5` becomes a command line option, `-q5` when the wrapper invokes the Blast search.

There are a few issues with pseudo columns that you must bear in mind:
- If a user references a pseudo column in a select list the wrapper must return a value for the pseudo column, even if it is an input-only datum. NULL is a good choice.
- Not all relational operators are appropriate for a particular pseudo column. Frequently, only one operator ('=' or '<' or '<=') can be used. During query planning, the wrapper must detect an invalid operator and reject the entire query, as opposed to simply rejecting the predicate. In the latter case, the federated server will attempt to process the predicate itself which could produce an unexpected result.
- Pseudo columns that exhibit default behavior can interfere with Materialized Query Tables (MQTs.) Because the optimizer does not recognize the default behavior, it might use an MQT when it shouldn't, or the optimizer might fail to use the MQT when it should. Queries against nicknames with pseudo columns should specify all predicates when used in an MQT definition.

Related concepts:
- "Modeling data source capabilities with function templates" on page 31

Related tasks:
- "Default cost model for federated queries" on page 12

Designing for wrappers

How wrappers work with options

A wrapper acts on options information in several ways:
- A wrapper validates options information that is specified in the SQL statements that are submitted during registration.

- The federated server's system catalog is a holding place that the wrapper uses to store configuration information, in the form of option values, until the wrapper needs this information to process a query.
- It makes use of such information to perform tasks. For example, suppose that as a prerequisite for routing error messages from the data source to the user, the wrapper needs to know whether it should route all such messages or only those for severe errors. You could create an option to which your user could assign values that denote these alternatives.

In preparing a wrapper for the actions listed in the foregoing, you need to:
- Decide whether the wrapper needs information that it should store as values of new options
- Determine how the wrapper should validate the options information that SQL statements provide for the federated server's system catalog,

You must not define option names that begin with `DB2_`. These names are reserved for use by IBM®.

Storing options information that the wrapper needs in the federated server catalog

When you create an option for a wrapper, you need to decide how you are going to obtain the value for that option. You have three choices:
- specify the option and value in one of the SQL statements used in registration
- hardcode the option and value in the wrapper, and pass them to the federated server's system catalog as a supplement to the information specified in the statements
- obtain certain option values from the data source itself. This typically doesn't apply to wrapper options, but might for any of the other possibilities. For example, a server might have a generic API from which you could obtain information about its maintenance level. If the wrapper records this as an option value, the wrapper might be able to use it so as to avoid certain query constructs known to cause problems. You might also be able to obtain nickname options and column options from the data source.

It is advisable to allow users to specify option values via SQL statements. If your user doesn't supply a value, use a hardcoded value or a value obtained from the data source as a default. If your user is never going to change the value, it should be hardcoded into the wrapper and not defined as an option.

Related concepts:
- "Deciding on wrapper options" on page 33

Related tasks:
- "Defining the CREATE WRAPPER statement for the data source" on page 34

Deciding on wrapper options

In general, the wrapper object does not require options. Options at this level affect all of the servers defined within the wrapper. One possible option or set of options might control debugging and diagnostic output that is produced by the wrapper.

Related concepts:
- "How wrappers work with options" on page 32

Related tasks:
- "Defining the CREATE WRAPPER statement for the data source" on page 34

Defining the CREATE WRAPPER statement for the data source

Mapping the wrapper concept means understanding what the system requires to initialize the wrapper and data source client libraries.

The wrapper usually does not require configuration options. However, you can define configuration options if your wrapper has a use for them. The rare but expected need for this arises when a global parameter is required to initialize the data source client library. For example, you might have to specify a global National Language Support (NLS) locale or a global wrapper debug level.

In Java, to define the CREATE WRAPPER DDL statement for your data source, you need to know at least the name of your unfenced wrapper class.

For example, if your wrapper class is named my.package.MyUnfencedWrapper, a corresponding CREATE WRAPPER statement could be:

```
CREATE WRAPPER MyWrapper LIBRARY 'db2qgjava.dll' options
                                                 (UNFENCED_WRAPPER_CLASS
                                                  'my.package.MyUnfencedWrapper')
```

Related concepts:
- "Deciding on wrapper options" on page 33
- "How wrappers work with options" on page 32

Designing for servers

Deciding on server options

The following example illustrates how to decide on what server options to define.

Consider a wrapper for data sources that contain spreadsheets. Assume that the wrapper communicates with these data sources over TCP/IP. To connect to a specific data source, the wrapper needs to know the host name and the number of that data source's TCP/IP port. Accordingly, you include options for the host name and the port number in the same SQL statement that defines the data source's server name.

For example, suppose data source A has a host name of `Peter` and is available through port `40`. On your CREATE SERVER statement you might create two new options, HOSTNAME and PORT. Your user would assign the host name `Peter` to the HOSTNAME option, and assign the number `40` to the PORT option.

Related concepts:
- "How wrappers work with options" on page 32

Related tasks:
- "Defining the CREATE SERVER statement for the data source" on page 35

Defining the CREATE SERVER statement for the data source

Mapping the server means defining the CREATE SERVER statement that matches your data source.

You need to determine your data source construct that maps to the server concept. For example, consider a document management system. Within a single enterprise, there can be multiple independent document archives at various locations, each containing several document collections. An application must connect to a specific archive in order to search the collections it contains. For this data source, you can model each archive as a server in the federated system

Then you need to determine the configuration information needed to configure the wrapper so it can connect to and use the server. These are the options on the CREATE SERVER statement. For example, in the case of the document archive, the NODE option could be the name of the computer where the archive resides.

There are two other available data items on the CREATE SERVER statement. These data items are the server type and version. When you have a set of data sources with a large amount of common behavior, but some minor variations, the server type and version can be used to control the operation of the wrapper. This eliminates the need to have separate wrappers for each variation and version.

For example, the CREATE SERVER statement associated with the previous information for your data source could be:

```
CREATE SERVER Server1 TYPE TYPEA WRAPPER MyWrapper OPTIONS(NODE 'Mname_Orion')
```

Related concepts:

- "Deciding on user mapping options" on page 35

Related tasks:

- "Defining the CREATE WRAPPER statement for the data source" on page 34
- "Defining the CREATE USER MAPPING statement for the data source" on page 36

Designing for user mappings

Deciding on user mapping options

Data sources rarely require user mapping options beyond REMOTE_AUTHID and REMOTE_PASSWORD.

To facilitate security, the federated server will encode the value of any user mapping option named REMOTE_PASSWORD before storing it in the federated server system's catalog. The federated server will decode it again before returning it to the wrapper. Use of this option name for authentication information that needs the federated server to store it securely is highly recommended.

You must implement these options yourself as the default versions of the User subclasses do not support them.

For data sources on Windows®, the Windows domain could be necessary to make network connections and for authentication. This is one option that you should support. For example, to record domain information, your wrapper could define an option named REMOTE_DOMAIN.

Related concepts:
- “Deciding on wrapper options” on page 33
- “Deciding on server options” on page 34
- “Deciding on nickname and column options” on page 29

Related tasks:
- “Defining the CREATE USER MAPPING statement for the data source” on page 36

Defining the CREATE USER MAPPING statement for the data source

Mapping the user defines the CREATE USER MAPPING statement for your data source.

For example, if you were writing a wrapper for a file archive, you want there to be one federated user per archive user. You want to define the following configuration information:
- REMOTE_AUTHID: User’s userid.
- REMOTE_PASSWORD: User’s password.

The resulting CREATE USER MAPPING statement could look like:

```
CREATE USER MAPPING FOR simon SERVER myserver
OPTIONS (REMOTE_AUTHID 'joy',
   REMOTE_PASSWORD 'open1up')
```

Related concepts:
- “Deciding on user mapping options” on page 35

Related tasks:
- “Defining the CREATE WRAPPER statement for the data source” on page 34
- “Defining the CREATE SERVER statement for the data source” on page 35

Chapter 5. Determining the SQL constructs that the data source can accept

You must consider which SQL constructs, including mapped functions, your data source can evaluate. You must consider the following questions:

- What kinds of head expressions your data source can accept. A head expression is an expression found in a SELECT list?
- What kinds of predicates your data source can accept?
- What kinds of joins your data source can accept?
- What functions your data source can accept?

The following sections discuss these considerations in more detail.

Determining the head expressions that the data source can accept

The wrapper looks at each expression in the SELECT list, also called a head expression, of the query fragment that the federated server supplies. If the data source can process the expression, then the wrapper indicates this by including the head expression in the reply. If the data source can not process the entire head expression, then the wrapper rejects the head expression by excluding it from the reply.

There is one limitation: DB2 Information Integrator v8.2 pushes down only column references for head expressions.

Related tasks:

- "Determining the predicates that the data source can accept" on page 37
- "Determining the joins that the data source can accept" on page 38
- "Determining the functions that the data source can accept" on page 38

Determining the predicates that the data source can accept

The wrapper looks at each predicate expression of the query fragment the federated server supplies. If the data source can process the expression, then the wrapper indicates this by including the predicate expression in the reply. If the data source can not process the entire predicate expression, then the wrapper rejects the predicate expression by excluding it from the reply.

The data source semantics for any predicate accepted must match the federated server semantics. The wrapper cannot control whether the federated server chooses to push down certain predicates. Any predicate can be processed by either the remote data source or the federated server. If the predicate semantics differ between the two, final results can differ, depending on whether the predicate is pushed down or not.

Before passing a query fragment to the wrapper, the federated server rewrites the predicates into what is known as conjunctive normal form. The predicates are rearranged into a list of predicate expressions connected by the `AND` operator.

For example:

```
WHERE expr1 AND expr2 AND expr3 ...
```

`Expr1`, `expr2` and `expr3` could be complex predicate expressions including NOT, AND, and OR operators. The wrapper must accept or reject expressions at the top-level. For example, if the first predicate, `expr1`, is `col1 = 0 OR col2 > 3` and the data source can evaluate `col1 = 0` but not `col2 > 3`, the wrapper rejects the entire `expr1` predicate.

Related tasks:

- "Determining the head expressions that the data source can accept" on page 37
- "Determining the joins that the data source can accept" on page 38
- "Determining the functions that the data source can accept" on page 38

Determining the joins that the data source can accept

Join requests include two or more nicknames, along with one or more predicates that relate the nicknames together. The wrapper examines join requests and determines whether the data source can support the particular join needed.

Related tasks:

- "Determining the head expressions that the data source can accept" on page 37
- "Determining the predicates that the data source can accept" on page 37
- "Determining the functions that the data source can accept" on page 38

Determining the functions that the data source can accept

The expression that defines a predicate typically contains one or more function invocations. If the wrapper accepts a predicate that includes invocation of a function DB2 UDB implements, the semantics of the function the data source runs must be identical to those of the federated server.

The expression that defines a predicate typically contains one or more function invocations. The functions in a predicate can be functions or operators that DB2 UDB implements (for example, >, LIKE, +, and CONCAT) or they can be custom functions specific to your wrapper that you registered by defining a function template. If the wrapper accepts a predicate that includes an invocation of a function that DB2 UDB could also evaluate, the semantics of the function the data source runs must be identical to those of DB2 UDB

The semantics of any DB2 UDB function that the data source intends to support must be identical to those of DB2 UDB itself. This includes issues like collating sequence and handling of nulls.

Related tasks:

- "Determining the head expressions that the data source can accept" on page 37
- "Determining the predicates that the data source can accept" on page 37
- "Determining the joins that the data source can accept" on page 38

Chapter 6. Designing for error handling

In preparing to write a wrapper, you need to decide how the wrapper should report the errors that it encounters.

If you *receive* a non-zero return code from a DB2 UDB support function or DB2 UDB-implemented member function,

1. Recover and reset the return code, if possible.
2. Otherwise, clean up locally and return the same code to the caller.

If you *discover* a "new" error,

1. Find the closest code in the SQL Messages reference guide.
2. Determine the number of tokens you can use from the message text. The maximum message length is 70 - (nTokens - 1).
3. Report the error.
4. Clean up locally and return the code to the caller.

The Java API uses Java exceptions to handle errors and also provides a specialized class to throw an exception that is converted into a DB2 error message. For Example:

```
throw new WrapperException(-1822, "FQftc", new String[]
                           { Integer.toString(remoteCode),
                           serverName, remoteMsg} );
```

Table 9 list categories of errors, each one followed by information about the codes and messages appropriate to it.

Table 9. Categories of errors and the associated error messages

Event	Message number	Message content
Authentication failure	SQL1403	The user name or password supplied was not correct. Should be used when the data source supports authentication.
Authentication, user mapping, missing	SQL1827	No user mapping is defined from local authorization ID *auth-ID* to server *server-name*. Although the descriptive text for this message refers specifically to ALTER and DROP statements, this message is also used when a user mapping is required and one is not present.
Blank truncation	SQL1844	Trailing blanks for column *column-name* were truncated between the remote data source and the federated server. This is a warning message to be produced when trailing blanks are truncated on transfer of data from the remote data source to the server. In cases where the truncation is due to code page conversion, use SQL1580.

Table 9. Categories of errors and the associated error messages (continued)

Event	Message number	Message content
Blank truncation, code page, conversion, data truncation	SQL1580	Trailing blanks were truncated while performing conversion from code page *source-code-page* to code page *target-code-page*. The maximum size of the target area was *max-len*. The source string length was *source-len* and its hexadecimal representation was *string*. A warning to be used when blanks are truncated (Wrapper_Utilities::convert_codepage returns CP_CONV_BUFFER_SMALL and the remaining characters are blank.)
Code page, conversion, data truncation	SQL0334	Overflow occurred while performing conversion from code page *source* to code page *target*. The maximum size of the target area was *max-len*. The source string length was *source-len* and its hexadecimal representation was *string*. Used when truncation/overflow occurs in data conversion. This should be issued when Wrapper_Utilities::convert_codepage returns CP_CONV_BUFFER_SMALL.
Code page, conversion, invalid data	SQL0191	Error occurred because of a fragmented MBCS character. This message should be produced when Wrapper_Utilities::convert_codepage returns CP_CONV_DBCS_TRUNCATE.
Code page, conversion, not supported	SQL0332	
Column, data type, local, altering	SQL0270	Function not supported (reason code = *reason-code*). Code 65 says "altering the nickname local type from the current type to the specified type is not allowed." This can be used on an ALTER NICKNAME ALTER COLUMN LOCAL TYPE statement.
Column, data type, local, invalid length precision scale	SQL0604	The length, precision, or scale attribute for column, distinct type, structured type, attribute of structured type, function, or type mapping *data-item* is not valid. When validating a nickname column specification (for CREATE or ALTER), use this message if one of the length, precision or scale is bad. *Data-item* should be the column name.
Column, data type, local, not supported	SQL3324	Column *name* has a type of *type* which is not supported. Can be used for an invalid local type for a nickname column.
Column, operator, not allowed	SQL1843	The *operator-name* operator is not supported for the *nickname-name.column-name* nickname column. Use this if your wrapper limits what operators can be applied to nickname columns.
Column, remote, not found	SQL0205	Column or attribute name *name* is not defined in *object-name*. Use this when an option references either a local column or remote column or attribute and the target does not exist. This is more specific than SQL0204.
Columns, too many	SQL0680	Too many columns are specified for a table, view or table function. Useful if your data source does not support as many columns as DB2 UDB.

Table 9. Categories of errors and the associated error messages (continued)

Event	Message number	Message content
Communication, lost connection, TCP/IP, general error	SQL30081	A communication error has been detected. Communication protocol being used: *protocol*. Communication API being used: *interface*. Location where the error was detected: *location*. Communication function detecting the error: *function*. Protocol specific error code(s): *rc1*, *rc2*, *rc3*. Use this for any TCP/IP communications errors, other than gethostbyname() not finding resolving a name (see SQL1336.) This includes some special conditions that indicate that a connection has been dropped.
Communication, TCP/IP, host not found	SQL1336	The remote host *hostname* was not found. Use this when your TCP/IP gethostbyname() fails to return a host. See SQL30081 for other TCP/IP errors.
Conversion, numeric, out of range	SQL0405	The numeric literal *literal* is not valid because its value is out of range. Use this message when converting from a remote data value to a local one.
Conversion, numeric, overflow	SQL0413	Overflow occurred during numeric data type conversion. Use when converting and an overflow occurs.
Data truncation	SQL1844, SQL1845	Data for column column-name was truncated between the remote data source and the federated server. This is the error message that corresponds to the SQL1844 warning. If truncation is due to code page conversion, use SQL0334.
Data source error	SQL1822	Unexpected error code *error-code* received from data source *data-source-name*. Associated text and tokens are *tokens*. As the message says, this is an error reported by the data source (for which there isn't a corresponding DB2 UDB message.) It should not be used for internal errors, nor for errors detected by the wrapper itself.
Datetime, invalid syntax	SQL0180	The string representation of a datetime value is incorrect. Use this one when converting from a remote source string to a DB2 Information Integrator date, time or timestamp value.
Datetime, out of range	SQL0181	The string representation of a datetime value is out of range. This message is the same as SQL0180.
Environment variable, missing	SQL5182	A required environment variable *variable-name* has not been set. This message should be issued when a variable is required in db2dj.ini and isn't there.
Name, column, undefined	SQL0204	*name* is an undefined name. Use this message when you have an option (such as REMOTE_TABLE) and the referenced object doesn't exist. See also SQL0205 for references to columns or attributes.
Name, column, undefined	SQL0205	Column or attribute name *name* is not defined in *object-name*. Use this when an option references either a local column or remote column or attribute and the target does not exist.
Option, add, invalid	SQL1840	The *option-type* option *option-name* cannot be added to the *object-type* object. Use this one for options that a wrapper might generate for its own use, or for options that can only be added at CREATE time and not ALTER.

Table 9. Categories of errors and the associated error messages (continued)

Event	Message number	Message content
Option, conflict	SQL1846	The *option-type-1* option *option-name-1* for the *object-name-1* object conflicts with *option-type-2* option *option-name-2* for the *object-name-2* object Use this when two (or more) options or option values conflict. Note that this could be a conflict between object types (such as a column option that is not valid with a particular nickname option.)
Option, drop, invalid	SQL1837	The required option *option-name* of type *option-type* cannot be for *object-name* cannot be dropped. Issued when the user attempts to DROP an option that cannot be dropped.
Option, duplicate	SQl1884, SQL1885	You specified *option-name* (an *option-type* option) more than once for *object-name*. Some options, like certain column options, can only be specified once. As an example, the PRIMARY_KEY column option for an XML wrapper nickname can only be specified on one column. In this context, the *object-name* would be the nickname.
Option, invalid	SQL1881	*option-name* is not a valid *option-type* option for *object-name*. Invalid option. Can be used if the option is not recognized at all, or can be used if the option is not valid in the current context, although the latter situation might be better handled by SQL1884.
Option, invalid value	SQL1882, SQL1842	The *option-type* option *option-name* cannot be set to *option-value* for *object-name*. Invalid option value. If the value is conflicting with another option value, use SQL1846N instead. If the value is a reference to some other entity (like a remote object, or even a local catalog object), use SQL0204.
Option, missing	SQL1883	*option-name* is a required *option-type* option for *object-name*. A required option was not specified. Use this for CREATE. For DROP, use SQL1837.
Option, set server option, invalid	SQL1841	The value of the *option-type* option *option-name* cannot be changed for the *object-name* object. Similar to SQL1840, but for SET.
Option, undefined	SQL1886	The *operation-type* operation is not valid because the *option-type* option *option-name* has not been defined for *object-name*. The DDL process detects this when the user tries to SET or DROP an option that is not defined.
Server, type, not valid	SQL1816	Wrapper *wrapper-name* cannot be used to access the *type-or-version* of data source (*server-type server-version*) that you are trying to define to the federated database. Used when validating the server type or version in the CREATE SERVER and ALTER SERVER statements. The component framework can handle this for you.
Server, version, not valid	SQL1817	The CREATE SERVER statement does not identify the *type-or-version* of data source that you want defined to the federated database. Used when type or version is required on the CREATE SERVER statement but not specified. The component infrastructure can handle this for you.

Table 9. Categories of errors and the associated error messages (continued)

Event	Message number	Message content
SQL, unsupported	SQL0142	The SQL statement is not supported. The problem is that there is no further information, which makes it very hard for a user to understand. SQL30090N with either one of the predefined reason codes, or text in place of the reason code is more useful.
User id, missing	SQL1027	

Related concepts:

- "Deciding on wrapper options" on page 33
- "Deciding on server options" on page 34
- "Deciding on nickname and column options" on page 29
- "Deciding on user mapping options" on page 35

Part 3. Developing and documenting wrappers

This part of the book takes you through the following tasks required to develop and document a wrapper:

- Coding your wrapper based on your design.
- Documenting your wrapper so users can quickly learn how to use it.

Chapter 7. Overview of data flows

Federated query processing and the objects that are involved

The following sections provide a detailed outline of a typical query flow and notes on the life cycles of various objects that are involved in query processing.

Typical flow of a federated query

The following list outlines the typical flow of a query. `XX` in the subclass names represents your particular wrapper.

1. Client application submits query that requires access to external data source.
2. The federated server determines (based on nicknames in query) which wrappers are relevant to this query. The following steps 3-12 apply to each such wrapper.
3. The federated server loads the unfenced generic wrapper library (C++) or class (Java), and invokes the bootstrapping entry point (hook function). The bootstrap function is only available in C++.
4. The federated server creates an instance of the wrapper's customized subclass of the unfenced generic wrapper class. Specifically:

   C++: The federated server invokes the bootstrapping entry point (hook function), which then creates an instance of wrapper's Unfenced_XX_Wrapper class.

   Java™: The federated server invokes the UnfencedXXWrapper class's constructor, which then creates an instance of the class.
5. The federated server invokes initialization method on new wrapper object.
6. The federated server determines which of the wrapper's servers are involved in the query (based on nicknames). The following steps 7-12 apply to each such server.
7. The federated server invokes the server-creation method on the wrapper object, to create an instance of the wrapper's server subclass.

   C++: Unfenced_XX_Server (subclass of Unfenced_Generic_Server)

   Java: UnfencedXXServer (subclass of UnfencedGenericServer)
8. The federated server invokes initialization method on the new server object.
9. The federated server submits Request objects that contains the query fragment to the wrapper via query planning method on server objects. Instances of the wrapper's nickname subclass will be created in the process.
10. The wrapper analyzes query fragments and returns one or more Reply objects to the federated server. Each Reply object contains an accepted query fragment, cost estimates, and an execution descriptor. Repeat steps 9–12 for each query fragment generated by the federated server optimizer.
11. The federated server invokes the selectivity-estimation method on the server object. It passes a list containing all the predicates from the query fragment that were not accepted by the wrapper.
12. The wrapper estimates the predicates' combined selectivity, using a custom selectivity-estimation method if one was supplied, or the default method otherwise.

13. The federated server optimizer chooses plan for entire query.
14. Repeat steps 2-8, substituting fenced for unfenced. In other words, create and initialize all necessary Fenced_XX_Wrapper and Fenced_XX_Server objects (FencedXXWrapper and FencedXXServer in Java). The following steps 15-25 apply to each server.
15. The federated server invokes the user-creation routine on the server object. It does this to create an instance of the wrapper's user subclass to represent the currently connected user.

    C++: Fenced_XX_User (subclass of Fenced_Generic_User)

    Java: FencedXXRemoteUser (subclass of FencedGenericRemoteUser)
16. The federated server invokes the initialization method on the new user object.
17. The federated server invokes the connection-creation method on the server object to create an instance of the wrapper's remote connection subclass.

    C++: Remote_Connection (XX_Connection)

    Java: RemoteConnection (XXConnection)
18. The federated server invokes connection method on the connection object. The wrapper connects to remote source (if necessary).
19. The federated server invokes query-creation method on the connection object to create an instance of wrapper's remote query subclass.

    C++: Remote_Query (XX_Query)

    Java: RemoteQuery (XXQuery)
20. The federated server invokes query-initiation method on the query object. The wrapper submits query to remote source, using execution descriptor from Reply (obtained from the query object).
21. The federated server invokes fetch method on the query object to request a result row from wrapper. The wrapper returns a result row to the federated server.
22. Repeat step 21 until no more rows to return.
23. If the federated server wants to re-execute the query, perhaps with an altered parameter value, the federated server repeats steps 20–22.
24. The federated server invokes the query-cleanup method on the query object. The wrapper cleans up cursors or other query-related resources.
25. The federated server destroys the query object.
26. Application terminates unit of work.
27. The federated server invokes commit/abort method on the connection object. This step is repeated for each server.
28. Application disconnects from database.
29. The federated server invokes connection-cleanup method on the connection object. The federated server closes the connection to the data source.
30. The federated server destroys the connection object. Steps 29 and 30 are repeated for each server.
31. Any instantiated Fenced/Unfenced wrapper, server, user, or nickname objects are destroyed.

Related concepts:
- "Control flow for query execution" on page 57
- "Communication between wrappers and foreign servers" on page 58
- "Control flow for query planning" on page 55

Related tasks:
- "Control flow for registration" on page 49
- "Control flow for initialization" on page 55

Related reference:
- "Life cycles of objects that are involved in federated queries" on page 49

Life cycles of objects that are involved in federated queries

- Fenced or Unfenced wrapper, server, user or nickname objects will not be created if they already exist. That is, if they have already been created in order to plan or execute a query submitted previously by the application.
- The federated server destroys Fenced or Unfenced wrapper, server, user or nickname objects associated with an application when they are not in use by an in-progress unit of work. If this occurs, the federated server creates objects as needed when the application submits another query. The usual reason for destroying such objects while the application remains connected to the database is because an application has issued an ALTER or DROP DDL statement. The ALTER or DROP DDL statements alter the properties of the wrapper, server, user or nickname that the object represents.
- The federated server will not clean up a connection to a remote data source or destroy an XX_Connection object immediately when it completes a unit of work. The connection will be left open for some time in case the application wants to execute another UOW that accesses the same remote source. The federated server will initiate cleanup after the application completes some number of units of work that do not reference the source, or when the application disconnects from the database.

Related concepts:
- "Communication between wrappers and foreign servers" on page 58
- "Typical flow of a federated query" on page 47

Control flows for processes

The following sections describe the control flows for the following processes:
- Registration
- Initialization
- Query planning
- Query execution

Control flow for registration

This topic introduces classes that model information that is stored in the federated server's system catalog when data sources and associated constructs are registered. It also outlines the generic control flows for the registration process and how to validate your wrapper's registration process.

Standard and options information

When the federated server registers a construct, one or both of two kinds of information go into the federated server's system catalog. *Standard information* consists of specifications that are common to most constructs of a given type. For example, when registering a server, you supply a user-assigned name for the server and the name of the wrapper that will be used to access the server on the CREATE SERVER statement. Because every server definition requires these pieces of information, a user-assigned name and the name of an associated wrapper can be regarded as standard information for servers.

In addition to standard information that the federated server maintains for all instances of a construct, the federated server's system catalog contains information whose nature varies from one type of data source to another. For example, IBM supplies a wrapper that accesses information stored in certain types of files. To access a file, the wrapper must know the file's name. Information like the filename is specific to this kind of data source. It would not make sense, for example, when accessing an Oracle source. The federated server stores such information in the catalogs by assigning values to variables called options. You specify option names and values using DDL. Thus, the file wrapper supports an option variable for nicknames, FILE_PATH, whose value you can specify with the CREATE NICKNAME statement.

Classes for both standard and options information

Several classes model all information—both standard and options—about data sources and associated constructs. These classes collectively are referred to as *construct information classes.* They are:

Table 10. The construct information classes in C++ and Java

C++	Java	Description
Wrapper_Info	WrapperInfo	models information about a wrapper
Server_Info	ServerInfo	models information about a data source
Nickname_Info	NicknameInfo	models information about a collection of data (for example a table or view) in a data source
Column_Info	ColumnInfo	models information about columns of values (or such column equivalents) within a collection of data at a data source
User_Info	UserInfo	models information about an authorization to use a data source

How objects of construct information classes are used

The federated server and wrappers use objects of construct information classes in the following ways:

- They serve as containers for the information that the system transfers from SQL statements and the wrapper to the federated server's system catalog. For example, a wrapper validates information in a CREATE SERVER statement and the federated server catalogs this information, along with any supplementary information that the wrapper provides. The system stores both the validated and supplementary information to the federated server's system catalog in a Server_Info object (ServerInfo in Java).

- They serve as containers for information as it is transferred from the federated server's system catalog to objects that represent constructs; for example, to a Server object that represents a specific data source. When the system makes the transfer, the receiving object initializes itself with this information.

  Each wrapper obtains authorization information from the federated server's system catalog in order to connect to a data source. The system passes this information to the wrapper in a User_Info object (UserInfo in Java). Similarly, the wrapper obtains cataloged information about the nicknames a query references, so that the wrapper can identify the corresponding data collections at the data source. The system passes this information to the wrapper in Nickname_Info objects (NicknameInfo in Java).

Flow for CREATE WRAPPER, CREATE SERVER, and CREATE USER MAPPING

The following topic uses the C++ class and method names. If you are developing your wrapper in Java, substitute the corresponding Java method name.

1. Application submits a CREATE YY DDL statement, where YY = WRAPPER, SERVER, or USER MAPPING.
2. The federated server parses the DDL statement, checks basic syntactic correctness, checks for a preexisting entity with the same name, and determines which wrapper is responsible for the entity to be created.
3. The federated server creates and initializes any parent objects needed to create a new entity of this type. To create a new User Mapping, the relevant wrapper and server objects must be instantiated first.

Table 11. The wrapper and server objects in C++ and Java

C++	Java
Unfenced_XX_Wrapper	UnfencedXXWrapper
Unfenced_XX_Server	UnfencedXXServer

4. The federated server creates a new instance of the relevant wrapper's server, user, or wrapper subclass. The federated server does this by calling the appropriate creation method on the parent object. The federated server does not call the appropriate creation method on the parent object on C++ wrappers that are created by loading the appropriate library and calling the hook function. For Java wrappers, the unfenced wrapper subclass is loaded directly. This object will be an instance of the class Unfenced_XX_YY where XX is the name you specified for your wrapper and YY is the name you specified for the type of object being created. For example:

   C++: Unfenced_File_Wrapper

   Java: UnfencedFileWrapper
5. The federated server creates an instance of the information subclass corresponding to the entity being created. This object will be an instance of the class YY_Info, and contain all the information from the application's DDL statement.
6. The federated server invokes the creation-validation method on the Unfenced_XX_YY object, passing the information object to the wrapper for inspection.
7. The wrapper checks the information from the DDL statement for consistency and correctness by inspecting the YY_Info object. In particular, the wrapper checks that option values are present where mandatory, valid, and mutually consistent.

8. The wrapper optionally constructs another YY_Info object, referred to as a delta-info object. The wrapper can assign values to options or control other aspects of the entity being created by providing information in the delta-info object that overrides or augments the information specified via the DDL statement.
9. The federated server merges the information from the DDL statement with the information from the delta-info object (if any), and stores the resulting configuration in the federated server's system catalogs.
10. The federated server invokes the initialization method for the Unfenced_XX_YY object.

Flow for CREATE NICKNAME

The following topic uses the C++ class and method names. If you are developing your wrapper in Java, substitute the corresponding Java method name.

Registration flow for CREATE NICKNAME adds an extra validation step that utilizes the creation-validation method on the nickname class. Since the fenced classes have the machinery for contacting the data source, this allows the wrapper to connect to the data source to collect configuration information not explicitly specified in the DDL.

1. Application submits a CREATE NICKNAME DDL statement.
2. The federated server creates the appropriate Unfenced_XX_Wrapper and Unfenced_XX_Server parent objects, and the Unfenced_XX_Nickname object representing the new nickname.
3. The federated server creates Fenced_XX_Wrapper, Fenced_XX_Server, and Fenced_XX_Nickname objects.
4. The federated server creates a Nickname_Info object from the information on the DDL statement.
5. The federated server invokes the creation-validation method on the Fenced_XX_Nickname object, passing the Nickname_Info object for inspection.
6. Wrapper optionally connects to the external source and obtains additional information from the data source. This information includes schema information, statistical information, or both.
7. Wrapper optionally creates a delta Nickname_Info object, containing information (possibly obtained from the data source) that augments or overrides the DDL.
8. The federated server merges information from the DDL statement with information from the delta Nickname_Info object (if any).
9. The merged Nickname_Info created by the Fenced_XX_Nickname validation method becomes the input to the Unfenced_XX_Nickname validation method, and the rest of the steps are the same as steps 6 through 10 of the wrapper, server, and user mapping registration flow.

Flow for ALTER WRAPPER, ALTER SERVER, ALTER NICKNAME, ALTER USER MAPPING

The following topic uses the C++ class and method names. If you are developing your wrapper in Java, substitute the corresponding Java method name.

The ALTER flow differs from the CREATE flow because the wrapper must determine whether new option values specified via DDL are consistent with current values that remain unchanged, as well as with each other.

1. Application submits one of the DDL statements listed previously for entity YY, where YY is one of Wrapper, Server, or User Mapping.
2. The federated server parses the DDL statement, checks basic syntactic correctness, checks to ensure an entity with the specified name exists, and determines which wrapper is responsible for the entity to be altered.
3. The federated server creates and initializes any parent objects needed to instantiate the object representing the entity to be altered. To instantiate an Unfenced_XX_User or Unfenced_XX_Nickname, the relevant Unfenced_XX_Wrapper and Unfenced_XX_Server objects must be instantiated first. The federated server will not create objects if they already exist.
4. The federated server creates a new instance of the relevant wrapper objects Unfenced_Generic_Wrapper, Unfenced_Generic_Server or Unfenced_Generic_User subclass, depending on the entity your user wants to alter. The federated server does this by calling the appropriate creation method on the parent object, except in the case of wrappers which are created by loading the appropriate library and calling the hook function. This object will be an instance of the class Unfenced_XX_YY where XX is the wrapper and YY is the type of object being created (for example Unfenced_File_Wrapper, Unfenced_Blast_Server, and so forth). The federated server will not create objects if they already exist. In Java, there is no library and no hook function. Instead, the federated server instantiates the appropriate wrapper subclass.
5. The federated server initializes the newly created Unfenced_XX_YY object.
6. The federated server creates an instance of the Catalog_Info subclass corresponding to the entity your user wants to alter. This object will be an instance of the class YY_Info (for example Server_Info, Remote_User_Info), and contain all the information from the application's DDL statement.
7. The federated server invokes the alter-validation method on the Unfenced_XX_YY object, passing the YY_Info object to the wrapper for inspection.
8. The wrapper checks the information from the DDL statement for consistency and correctness by inspecting the YY_Info object. In particular, the wrapper checks to ensure that new option values are consistent with existing ones whose values remain unchanged and that mandatory options are not dropped.
9. The wrapper optionally constructs another YY_Info object, referred to as a delta-info object. The wrapper can assign values to options or control other aspects of the entity being altered by providing information in the delta-info object that overrides or augments the information specified via the DDL statement.
10. The federated server merges the information from the DDL statement with the information from the delta-info object (if any), and stores the resulting configuration in the federated server's system catalogs.
11. The federated server destroys the Unfenced_XX_YY object, which was initialized with old catalog information that does not reflect this DDL statement. The federated server recreates it when it is next needed and initialize it with the updated information.

General steps for validating the registration process

Validation of your wrapper's registration process follows these general steps:

CREATE DDL

| The following topic uses the C++ class and method names. If you are developing
| your wrapper in Java, substitute the corresponding Java method name.

Validation for any of the CREATE DDL should be necessary only if the wrapper developer has defined specific options. The following is a generic procedure for validating your registration. See the following sections for the specific validation procedure for each wrapper building block.

1. For each Catalog_Option in the XX_Info object:
   a. Determine if it is a reserved DB2 UDB option by calling is_reserved_xx_option() (Java API provides the method isReserved() on the CatalogOption class that has similar functionality). If it is reserved, skip to the next one.
   b. Determine if it is an option supported by the wrapper; if it is not, issue an SQL1881N error message.
   c. Determine if the option value is valid for that option; if it is not, issue an SQL1882N error message.
2. If your user omits any required options, issue an SQL1883N error message.
3. Determine that all supplied option values are consistent with each other (as applicable.) Use the SQL1882N error message to report invalid or inconsistent values.
4. If necessary, provide additional options:
   a. Determine if a delta XX_Info object has been allocated. Allocate one if needed.
   b. Use the add_option method of XX_Info to add new options as necessary.

ALTER DDL

Like CREATE, this is only necessary if the wrapper developer has defined wrapper-specific options. ALTER processing differs from CREATE processing in two significant ways: First, the validation routine must take into consideration the current state of the wrapper. The information supplied in the XX_Info object will only be that information which is new or changing. Second, the validation routine must pay attention to the action parameter for each option. Most importantly, the validation routine must issue an SQL1881N error message if the user attempts to DROP a required option. Also, attempting to check the value of an option that is being DROPped will result in problems such as the engine crashing with an addressing error.

Related concepts:
- "Control flow for query execution" on page 57
- "Communication between wrappers and foreign servers" on page 58
- "Control flow for query planning" on page 55
- "Typical flow of a federated query" on page 47

Related tasks:
- Chapter 6, "Designing for error handling," on page 39
- "Control flow for initialization" on page 55
- "Altering a wrapper" in the *Federated Systems Guide*

Related reference:
- "ALTER WRAPPER statement" in the *SQL Reference, Volume 2*

Control flow for initialization

| The following outlines the control flow for the initialization process.

This outline uses the C++ object names. If you are developing your wrapper in Java, substitute the corresponding Java object name.

This flow is truly generic, applying to all objects that are instances of a subclass of a Fenced_Generic or Unfenced_Generic base class (for example Unfenced_Blast_Nickname, Fenced_File_User, and so forth).

1. The federated server creates an instance of a wrapper-defined subclass, "X" (for example Fenced_Excel_Server), using the appropriate creation method on its parent object (or, for wrappers, the hook function).
2. The federated server creates a YY_Info (for example Server_Info) object containing all known information about the entity that the object represents, as stored in the appropriate the federated server's system catalogs. All Catalog_Info objects include a list of Catalog_Option objects, which supply the names and values of any options defined for the corresponding entity. Nickname_Info objects include a list of Column_Info objects, one for each column of the nickname.
3. The federated server invokes the initialization method on the "X" object created in step 1, passing the YY_Info object created in step 2.
4. The wrapper optionally extracts information (for example option values) from the YY_Info object, and stores it in member variables of the "X" object for more efficient access. The federated server copies and attaches the corresponding YY_Info object to the "X" object. For Nickname objects (fenced or unfenced), the federated server does not copy the Nickname_Info object.

Related concepts:
- "Control flow for query execution" on page 57
- "Communication between wrappers and foreign servers" on page 58
- "Control flow for query planning" on page 55
- "Typical flow of a federated query" on page 47

Related tasks:
- "Control flow for registration" on page 49

Control flow for query planning

The following outlines the flow for the query planning process.

This outline uses C++ object names. If you are developing your wrapper in Java™, substitute the corresponding Java object name.

The flow outline assumes that the federated server is asking the wrapper to plan a single-table query fragment with one or more head expressions and predicates. The federated server plans single-table fragments first, followed by fragments that join two nicknames from the source, then three-way joins, and so forth For these more complex fragments, there will be additional quantifier handles in the Request and some of the predicate handles in the Request will represent join predicates.

1. The federated server creates a Request object describing the query fragment, and invokes the planning method on the appropriate Unfenced_XX_Server object.

2. The wrapper creates a Reply object for this request. If the wrapper supports a custom cost model, this will be an instance of a Reply subclass defined by the wrapper. If the wrapper uses the default cost model, the wrapper instantiates the base Reply class.
3. The wrapper obtains a handle for the quantifier representing the nickname over which the query fragment ranges (The Java API uses instances rather than handles).
4. Using the handle, the wrapper asks the federated server for an instance of its Unfenced_Generic_Nickname subclass, Unfenced_XX_Nickname, corresponding to the nickname found in the query fragment.
5. The federated server creates and initializes an Unfenced_XX_Nickname object.
6. As necessary, wrapper obtains information about nickname columns, configuration, and so forth from Unfenced_XX_Nickname object.
7. The wrapper adds handle for quantifier to Reply.
8. The wrapper obtains a handle for one of the head expressions in the query fragment.
9. The wrapper uses a handle to obtain a Request_Exp object that describes the head expression.
10. The wrapper determines whether data source can compute the value of the head expression that is represented by Request_Exp. Wrapper repeats step 12 recursively descending the Request_Exp tree until a the wrapper makes a decision.
11. If the data source can compute the entire head expression, the wrapper adds the handle for the head expression to the Reply object.
12. Repeat steps 9-12 for each additional head expression in the request.
13. The wrapper obtains a handle for one of the predicates in the query fragment.
14. The wrapper uses the handle to obtain a Request_Exp object that describes the predicate.
15. The wrapper determines whether the data source can process the entire predicate represented by the Request_Exp. The wrapper will need to recursively descend the Request_Exp tree to make this decision.
16. If the data source can process the predicate, the wrapper adds the handle for the predicate to the Reply object.
17. Repeat steps 15-17 for each additional predicate in the request.
18. If the data source returns rows in a particular order that is compatible with DB2® Information Integrator collation, the wrapper adds order descriptions to the Reply.
19. The wrapper examines query sub-fragment that is represented by Reply and generates persistent representation to store in wrapper execution descriptor.
20. The wrapper stores execution descriptor in Reply.
21. If the wrapper wishes to generate alternative plans for this query fragment, repeat steps 9-23 for each alternative.
22. The wrapper returns list of Reply objects to the federated server.
23. The federated server invokes costing methods on Reply object to calculate cost and cardinality for corresponding wrapper plan.
24. If wrapper uses a custom cost model, wrapper's costing methods calculate cost and cardinality, optionally referring to custom statistics that are obtained from the Unfenced_XX_Nickname object. Otherwise, default implementations will calculate these values by using the default cost model.
25. Repeat steps 26-27 for each additional Reply object, if any.

Related concepts:

- "Control flow for query execution" on page 57
- "Communication between wrappers and foreign servers" on page 58
- "Typical flow of a federated query" on page 47

Related tasks:

- "Control flow for registration" on page 49
- "Control flow for initialization" on page 55

Control flow for query execution

| The following outlines the query execution process.

This query uses C++ object names. If you are developing your wrapper in Java™, substitute the corresponding Java object name.

1. The federated server invokes query-initiation method on instance of wrapper's Remote_Query subclass, XX_Query.
2. Wrapper obtains wrapper execution descriptor from XX_Query object. Using information in the descriptor, the wrapper prepares to execute the query fragment at the remote source.
3. Wrapper obtains output Runtime_Data_List from XX_Query object and determines mapping between data that are returned from data source and data the wrapper returns to the federated server.
4. If query fragment had parameters, wrapper obtains parameter values from corresponding elements of the input Runtime_Data_List, obtained from the XX_Query object.
5. Wrapper converts parameter values (if any) from SQL (DB2 UDB) representation to the representation expected by the data source.
6. Wrapper initiates execution of query fragment at remote source.
7. The federated server invokes fetch method on XX_Query object.
8. Wrapper retrieves at least one row's worth of data from data source. The wrapper buffers additional rows.
9. The wrapper converts result values from the representation provided by the data source to the SQL representation expected by the federated server . It copies each converted value to a DB2® buffer by invoking a method on the appropriate element of the output Runtime_Data_List.
10. Repeat steps 7-9 until the data source has no more rows to return.
11. The federated server invokes query cleanup method XX_Query. Status flag indicates whether DB2 will re-execute the query fragment as part of the same application query.
12. If query is re-executed, the federated server invokes a query re-execution method on XX_Query. Steps 4-11 are repeated until no more re-executions are requested by the federated server.

Related concepts:

- "Communication between wrappers and foreign servers" on page 58
- "Control flow for query planning" on page 55
- "Typical flow of a federated query" on page 47

Related tasks:

- "Control flow for registration" on page 49
- "Control flow for initialization" on page 55

Communication between wrappers and foreign servers

Some of the ways in which a wrapper participates in query processing are to translate query fragments into requests that the data source understands, to submit these subqueries to the data source, and to retrieve rows of results that the data source returns. Methods of the unfenced generic server subclass plan these tasks. Methods of the remote query subclass perform the execution; and you have considerable discretion in determining how to distribute the tasks among the methods. To illustrate:

Example 1

The following example begins when a wrapper for a file server starts processing a SELECT statement in a query fragment that references columns and contains no predicates.

1. During query planning, the optimizer examines a query fragment and sends the wrapper a request based on the fragment. The wrapper examines the request and returns a reply along with a wrapper plan that lets the optimizer know what the wrapper can and cannot do with the data source. The optimizer creates a query plan for execution that includes the wrapper plan.
2. During query execution, the optimizer sends the wrapper plan to the wrapper.
3. The wrapper executes the plan against the data source.
4. On the basis of the value of an option that was supplied by the wrapper execution descriptor, the wrapper determines the name and location of the data source file that contains the data set that corresponds to the nickname.
5. The wrapper examines the wrapper plan to determine the names of the columns to be retrieved.
6. On the basis of the value of an option that was supplied by the wrapper execution descriptor, the wrapper determines which ordinal column in the file corresponds to each named column in the query.
7. On the basis of the value of an option that was supplied by the wrapper execution descriptor, the wrapper determines the terminating delimiter for each column named in the query.
8. The wrapper opens the file and begins to read data. It uses information determined previously to parse and format rows to return to the federated server.

Example 2

The following example begins when a wrapper for a document server starts processing a SELECT statement in a query fragment. This fragment references three kinds of data source constructs: repositories of documents, constructs that are called *attributes*, and a text-search function. In the federated server's system catalog, the federated server defines the repositories as DB2® UDB tables, DB2 UDB defines the attributes as columns of these tables, and the function maps to a function template.

1. During query planning, the optimizer examines a query fragment and sends the wrapper a request based on the fragment. The wrapper examines the request and returns a reply along with a wrapper plan

that lets the optimizer know what the wrapper can and cannot do with the data source. The optimizer creates a query plan for execution that includes the wrapper plan.

2. During query execution, the optimizer sends the wrapper plan to the wrapper.
3. The wrapper executes the plan against the data source.
4. On the basis of the value of an option that was supplied by the wrapper execution descriptor, the wrapper determines which repository of documents corresponds to the nickname.
5. The wrapper examines the wrapper plan to determine the names of the columns to be retrieved.
6. On the basis of the value of an option that was supplied by the wrapper execution descriptor, the wrapper determines which attribute corresponds to each column named in the query.
7. The wrapper identifies a text-search function in the wrapper execution descriptor.
8. Using information determined previously (for example, the names of the repositories of documents, the names of the attributes and the text-search predicate), the wrapper generates a statement in the query language of the document server.
9. The wrapper submits the query to the document server's API, and begins to read data to return to the federated server.

Related concepts:

- "Control flow for query execution" on page 57
- "Control flow for query planning" on page 55
- "Typical flow of a federated query" on page 47

Related tasks:

- "Control flow for registration" on page 49
- "Control flow for initialization" on page 55

Related reference:

- "Life cycles of objects that are involved in federated queries" on page 49

Chapter 8. Developing with wrapper classes

The following sections describe how to develop a wrapper and the classes you need to use when developing a wrapper.

Typical procedure for developing a wrapper

This topic describes the typical procedure for developing a wrapper. Before you start to develop the wrapper, first:

- Familiarize yourself with the SQL statements used in registration, the base classes that you'll need to subclass, and the API of the data source.
- Determine how to model as relational tables the collections of data at the data source.
- Determine how to represent in the DB2® UDB catalog the metadata about collections of data at the data source.
- Perform other planning activities, as needed.

When you develop a wrapper, you typically perform the following steps:

1. Implement and test the wrapper subclasses:

   C++: Unfenced_Generic_Wrapper and Fenced_Generic_Wrapper

   Java™:
   UnfencedGenericWrapper and FencedGenericWrapper

   Test your implementation by running the CREATE WRAPPER statement. Examine the resulting entries made in the SYSCAT.WRAPPERS and SYSCAT.WRAPOPTIONS catalog views.
2. Implement and test the server classes:

   C++: Unfenced_Generic_Server and Fenced_Generic_Server

   Java: UnfencedGenericServer and FencedGenericServer

   Test your implementation by running the CREATE SERVER statement and examining the resulting entries made in the SYSCAT.SERVERS and SYSCAT.SERVEROPTIONS catalog views.
3. (Optional) Implement and test the user classes if the wrapper needs authorization and contacts the data source during nickname validation.

   C++: Unfenced_Generic_User and Fenced_Generic_User

   Java: UnfencedGenericRemoteUser and FencedGenericRemoteUser

   Test your implementation by running the CREATE USER MAPPING statement and examining the resulting entries made in the SYSCAT.USEROPTIONS catalog view.
4. (Optional) Implement the remote connection subclass if the wrapper contacts the data source during nickname validation.

   C++: Remote_Connection

   Java: RemoteConnection

5. If you want data sources to support passthru, implement the remote passthru class:

   C++: Remote_Passthru

   Java: RemotePassthru

   Test your implementation of both the remote passthru subclass and the remote connection class by running a passthru request that references collections of data in your data source.
6. Implement the nickname subclasses:

   C++: Unfenced_Generic_Nickname and Fenced_Generic_Nickname

   Java: UnfencedGeneric Nickname and FencedGeneric Nickname

   Test your implementation by running the CREATE NICKNAME statement for several collections of foreign server data, and examining the resulting entries in the SYSCAT.TABLES, SYSCAT.TABOPTIONS, SYSCAT.COLUMNS and SYSCAT.COLOPTIONS catalog views and the SYSTAT.TABLES and SYSTAT.COLUMNS catalog tables. If you want users to use a data source function that has no DB2 UDB equivalent, create a function template that corresponds to the function.
7. (Optional) Implement and test the user classes if the wrapper needs authorization and does not connect to the data source during nickname validation.

   C++: Unfenced_Generic_User and Fenced_Generic_User

   Java: UnfencedGenericRemoteUser and FencedGenericRemoteUser

   Test your implementation by running the CREATE USER MAPPING statement and examining the resulting entries made in the SYSCAT.USEROPTIONS catalog view.
8. If you did not do so in Step 4, implement the remote connection subclass.

   C++: Remote_Connection

   Java: RemoteConnection
9. Implement the remote query subclasses.

   C++: Remote_Query

   Java: RemoteQuery

   To test your implementation of both this class and the remote connection class, submit a federated query that references collections of data in your data source.
10. Document your wrapper.

Related concepts:
- "Tips for developing wrappers" on page 63

Related tasks:
- "Trusted and fenced mode process environments" on page 64
- "Wrapper classes" on page 69
- "Server classes" on page 72
- "User classes" on page 80

Implementations of subclasses and methods

In familiarizing yourself with the wrapper interface, consider:

- What base classes to use. Be aware that:
  - Table 12 shows the base classes that require you to create your own implementations of subclasses.

Table 12. Base classes that require implementations of subclasses

C++	**Java™**
Unfenced_Generic_Wrapper	UnfencedGenericWrapper
Fenced_Generic_Wrapper	FencedGenericWrapper
Unfenced_Generic_Server	UnfencedGenericServer
Fenced_Generic_Server	FencedGenericServer
Unfenced_Generic_Nickname	UnfencedGenericNickname
Fenced_Generic_Nickname	FencedGenericNickname
Unfenced_Generic_User	UnfencedGenericRemoteUser
Fenced_Generic_User	FencedGenericRemoteUser
Remote_Connection	RemoteConnection
Remote_Query	RemoteQuery

  - If you are implementing your own cost model, you need to subclass the following:
    - Reply
  - If your wrapper supports passthrough, you need to subclass:
    - Remote_Passthru (RemotePassthru in Java)
  - You need to use IBM®'s implementations of certain classes.
- What to include in the implementations that you create. Be aware that:
  - You must override the default implementations of certain methods with customized implementations of your own.
  - You have the choice of using certain default implementations of methods or overriding them with customized implementations of your own.
  - You can add new methods of your own. Note that when you want to do so, you can build new methods that call methods of the parent class that you are implementing.
  - You can add new data members.

Related tasks:

- "Reply class" on page 83
- "Remote passthru class" on page 97

Tips for developing wrappers

When you write a wrapper, be aware that:

- The Wrapper_Utilities class (WrapperUtilities in Java™) provides services to help you with your work. For example, if you run into a problem with your code, you can use the class's trace_data method. The trace_data method generates a record that can help you locate and identify the problem.

- You can produce the wrapper in several versions, the first one limited to basic services, and the subsequent versions progressively more functional and efficient.

Related concepts:
- "Implementations of subclasses and methods" on page 63
- "Typical procedure for developing a wrapper" on page 61

Trusted and fenced mode process environments

During query execution, the federated server can improve performance by creating multiple subagents (threads or processes) to access data sources in parallel. Code executing in the subagent environment can be isolated more easily. This isolation makes failures less likely to cause loss of integrity for the system. For this reason, the portion of the wrapper that is involved in query execution is separated from the piece that is involved in query planning.

As a result, wrappers are divided into two parts:
- The *unfenced* portion is involved in query planning.
- The *fenced* portion is involved in query execution.

Each of the wrapper building blocks must therefore be assigned to either the fenced or the unfenced portion of the wrapper. In some cases, the building blocks must be split between the two. If communication with the external source is required, the function goes in the fenced portion.

The processing environment varies slightly depending on whether the wrapper was developed in C++ or Java.

C++ Processing Environment

A C++ wrapper is linked as two separate libraries, one containing the class implementations for the unfenced portion of the wrapper and one containing the class implementations for the fenced portion. In what is known as *fenced mode*, the fenced portions of wrappers are loaded into special isolated subagents called the Fenced-Mode processes (FMP). Unfenced portions of wrappers are loaded into the main federated server agent. Figure 7 on page 65 illustrates the fenced-mode process model.

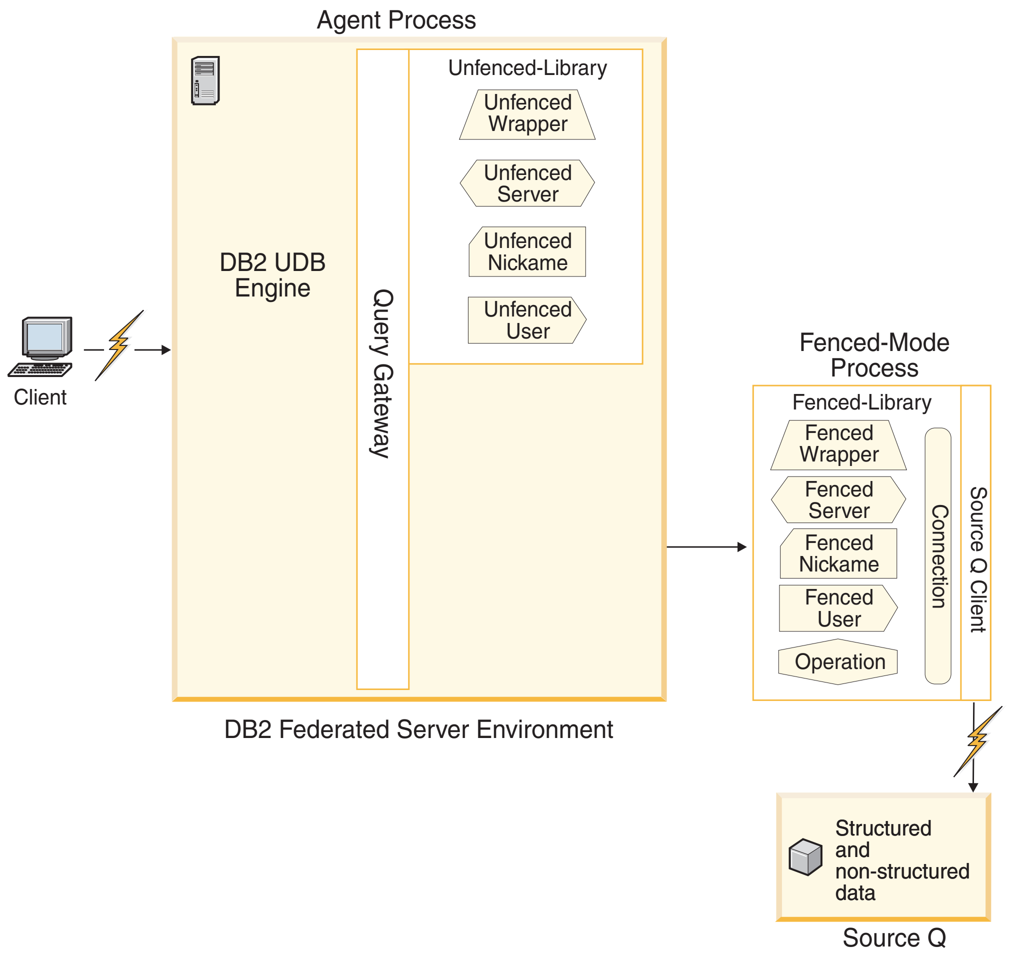

Figure 7. Fenced-mode process model for C++

Figure 7 shows the two separate libraries containing the fenced and unfenced portions of the wrapper and the separate areas where they are loaded; unfenced portions in the federated server environment and fenced portions in the fenced-mode process.

When parallelism is not an issue, performance will be better if you do not use subagents and all the function takes place within the federated server environment. Since you sacrifice the safety of isolation, this mode is called *trusted mode*. Figure 8 on page 66 illustrates the trusted-mode process model.

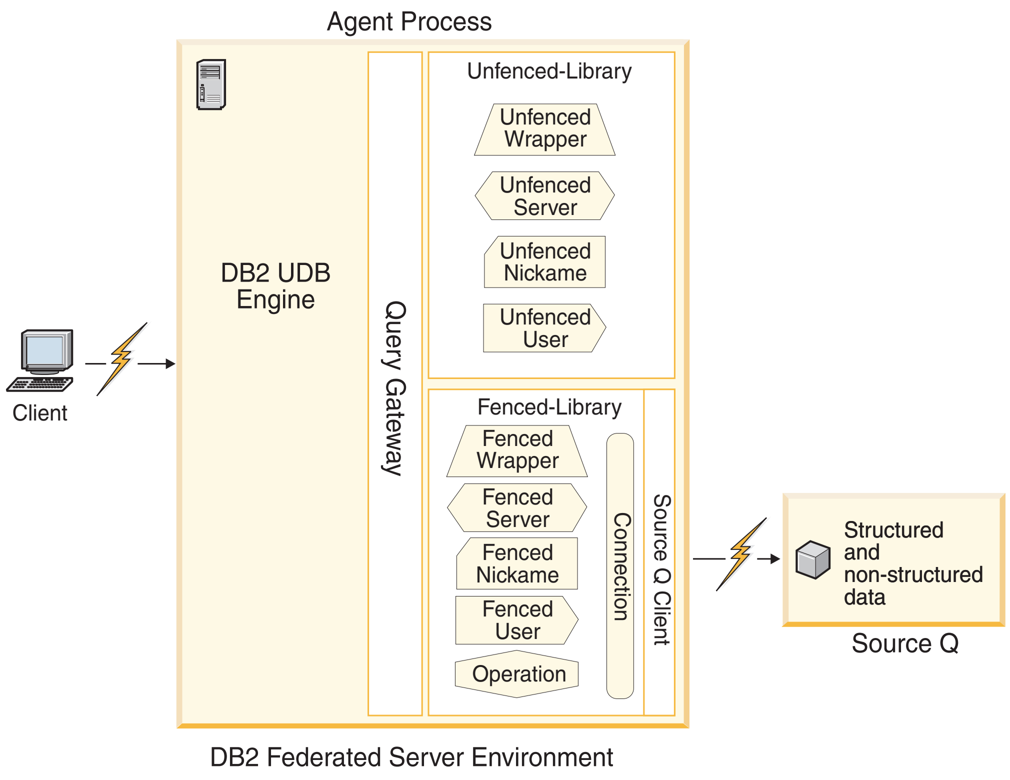

Figure 8. Trusted–mode process model for C++

Figure 8 shows that in trusted mode, the fenced and unfenced portions of the wrapper are both loaded into the federated server environment.

While DB2 Information Integrator only supports trusted-mode execution in V8.1, you must design wrappers to support both trusted-mode and fenced-mode execution.

Java Processing Environment

Wrappers that are developed in Java always run entirely in fenced-mode processes. Both the fenced and unfenced Java classes run in fenced-mode processes. Figure 9 on page 67 illustrates the fenced-mode processing environment for Java wrappers.

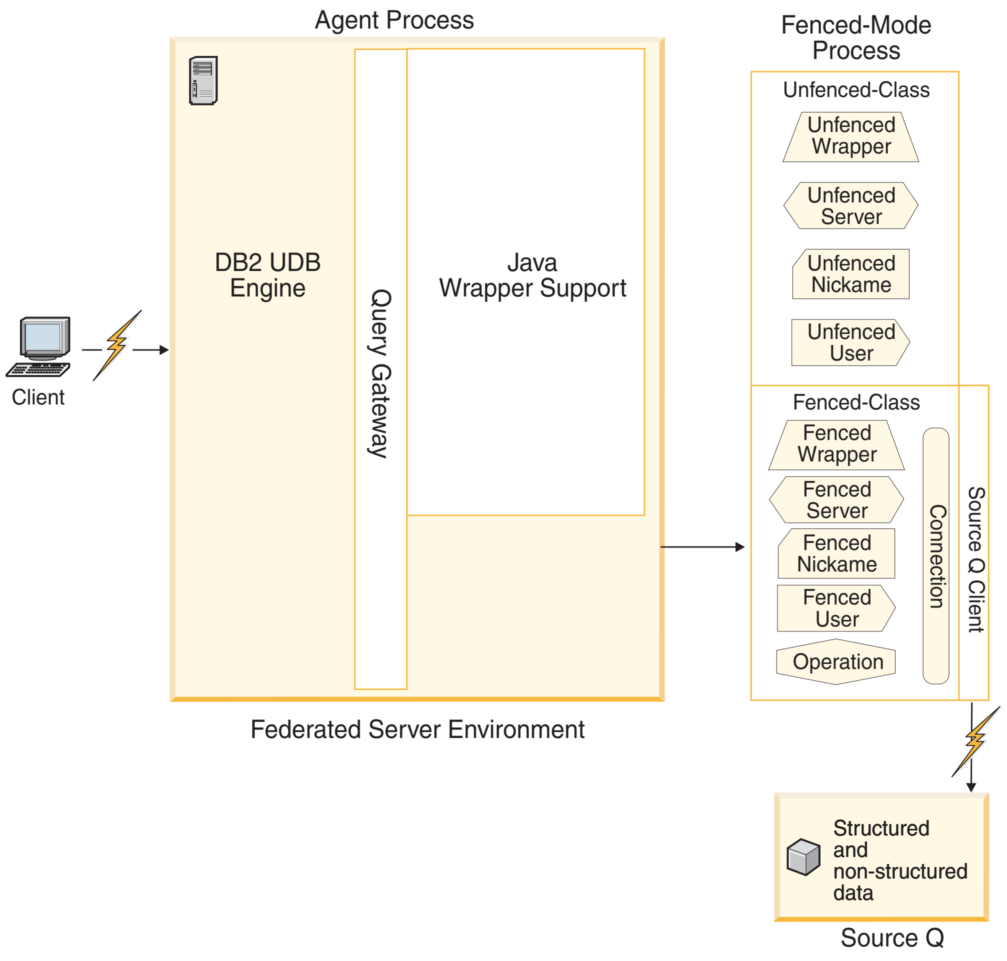

Figure 9. Fenced-mode process model for Java

| Figure 9 shows the fenced and unfenced classes running in fenced-mode processes,
| not in the federated server environment.

Related concepts:

- "Classes for communications between wrappers and data sources" on page 69
- "Implementations of subclasses and methods" on page 63
- "Typical procedure for developing a wrapper" on page 61
- "Tips for developing wrappers" on page 63

Related tasks:

- "Mapping parts of a wrapper to classes" on page 67

Mapping parts of a wrapper to classes

Many of the building blocks have unfenced and fenced 'sides' to them that you must code using the respective classes.

The unfenced side takes part in query planning. The fenced side is involved in the query execution phase to protect the federated server from the data source's environment when the federated server is in fenced-mode.

| *Table 13. The unfenced and fenced class of the basic wrapper building blocks in C++*

Building block	C++ unfenced class	C++ fenced class
Wrapper	Unfenced_Generic_Wrapper	Fenced_Generic_Wrapper
Server	Unfenced_Generic_Server	Fenced_Generic_Server
Nickname	Unfenced_Generic_Nickname	Fenced_Generic_Nickname
User	Unfenced_Generic_User	Fenced_Generic_User
Connection	none	Remote_Connection
Operation	none	Remote_Operation

| *Table 14. The unfenced and fenced class of the basic wrapper building blocks in Java*

Building block	Java unfenced class	Java fenced class
Wrapper	UnfencedGenericWrapper	FencedGenericWrapper
Server	UnfencedGenericServer	FencedGenericServer
Nickname	UnfencedGenericNickname	FencedGenericNickname
User	UnfencedGenericRemoteUser	FencedGenericRemoteUser
Connection	none	RemoteConnection
Operation	none	RemoteOperation

Related concepts:
- "Typical procedure for developing a wrapper" on page 61
- "Tips for developing wrappers" on page 63

Related tasks:
- "Trusted and fenced mode process environments" on page 64

Chapter 9. Classes for coding wrappers

This section describe the classes you use to code wrappers.

Classes for communications between wrappers and data sources

Table 15 shows the base classes that model the connection to the data source and the operation that the data source performs.

Table 15. Base classes that model the connection to the data source

Base classes in C++	Base classes in Java™
Remote_Query	RemoteQuery
Remote_Connection	RemoteConnection
Remote_Operation	RemoteOperation
Runtime_Data	RuntimeData
Runtime_Data_List	RuntimeDataList
Runtime_Data_Desc	RuntimeDataDesc
Runtime_Data_Desc_List	RuntimeDataDescList
Remote_Passthru	RemotePassthru

Like connections, data source operations are transient. Therefore, the federated server stores no information about them in the federated server's system catalog.

Related concepts:
- "Implementations of subclasses and methods" on page 63
- "Typical procedure for developing a wrapper" on page 61
- "Tips for developing wrappers" on page 63

Related tasks:
- "Trusted and fenced mode process environments" on page 64
- "Mapping parts of a wrapper to classes" on page 67
- "Remote connection class" on page 90
- "Runtime data classes" on page 94
- "Runtime data description classes" on page 96
- "Remote passthru class" on page 97

Related reference:
- "Remote query class" on page 92

Wrapper classes

The following sections describe the unfenced generic wrapper and fenced generic wrapper classes.

Unfenced_Generic_Wrapper class

C++: Invoking the function UnfencedWrapper_Hook in the shared library containing the wrapper code creates an instance of your unfenced wrapper subclass. The federated server calls this function when it first loads the wrapper's unfenced shared library. If a DDL operation changes information pertaining to a wrapper, the federated server will destroy and recreate the wrapper object before its next use.

Java: When the federated server loads the unfenced wrapper class, an instance of the class is automatically created. The name of the wrapper-specific unfenced wrapper class is specified as the value of the UNFENCED_WRAPPER_CLASS option in the CREATE WRAPPER statement.

The unfenced generic wrapper base class implementation maintains the following information:

- Wrapper name.
- Wrapper type: 'N' for nonrelational.
- Wrapper version: a wrapper-specified version number that represents the version of the wrapper code that was executing at the time the wrapper registered with the federated server. You can compare this value with the version of the currently executing code to assure compatibility.
- Top-level filename of wrapper module library.
- A wrapper information object containing all the information pertaining to this wrapper that was stored in the federated server's system catalog as a result of executing CREATE WRAPPER or ALTER WRAPPER DDL statements.

Only member functions should inspect or alter this information.

Required customization for all wrappers

The unfenced generic wrapper subclass must implement:

- A constructor, which calls the corresponding unfenced generic wrapper base class constructor.

  C++: Unfenced_Generic_Wrapper

  Java: UnfencedGenericWrapper
- create server: a method to create an instance of your unfenced generic server subclass.

  C++: Unfenced_Generic_Server

  Java: UnfencedGenericServer

Additional customization

- If you need to store additional information in instances of your unfenced generic wrapper subclass, override the default implementation of the initialize_my_wrapper() function to extract this information from the wrapper information object supplied as a parameter. In C++, you cannot retain a pointer to this wrapper information object. If you choose to refer to the information in this form, use the data member that contains a copy of the wrapper information object. In Java, the wrapper can retain a reference to the WrapperInfo object but there is no copy function.
- If you define wrapper options, override the default implementations of the verify_my_register_wrapper_info function and verify_my_alter_wrapper_info function to verify that the options and values supplied on the DDL are valid. If you wish to alter an option value supplied on the DDL, or supply additional

options, your implementation should supply the overriding/additional information via a “delta” wrapper information object. Before allocating a new “delta” object, check whether one already exists. If a “delta” object does exist, do not allocate another. In Java, the “delta” wrapper information object is actually the return object of the verifyMyRegisterWrapperInfo and verifyMyAlterWrapperInfo methods. This object must be created by the wrapper.
- If your unfenced generic wrapper subclass points to any out-of-line storage you have allocated, you must implement a destructor for your subclass which frees this storage.

Table 16. Virtual functions

Virtual function in C++	Virtual function in Java	Default behavior
initialize_my_wrapper	initializeMyWrapper	No-op
create_server	createServer	Error
verify_my_register_wrapper_info	verifyMyRegisterWrapperInfo	Verifies no non-DB2 Information Integrator options were specified
verify_my_alter_wrapper_info	verifyMyAlterWrapperInfo	Calls verify my register wrapper information object

Table 17. Public/protected member function

Public/protected member function in C++	Public/protected member function in Java	Behavior
is_reserved_wrapper_option	isReserved	Returns true if option handled by DB2 UDB

Fenced_Generic_Wrapper class

C++: Invoking the function FencedWrapper_Hook in the shared library containing the wrapper code creates an instance of your fenced wrapper subclass. The federated server calls this function when it first loads the wrapper’s fenced shared library. If a DDL operation changes information pertaining to a wrapper, the federated server will destroy and recreate the wrapper object before its next use.

Java: When the federated server loads the fenced wrapper class, an instance of the class is automatically created. The name of the wrapper-specific fenced wrapper class is specified as the value of the FENCED_WRAPPER_CLASS option in the CREATE WRAPPER statement.

The fenced generic wrapper base class implementation maintains the following information:
- Wrapper name.
- Wrapper type: ’N’ for nonrelational.
- Wrapper version: a wrapper-specified version number that represents the version of the wrapper code that was executing at the time the wrapper registered with the federated server. This version number is an integer, not a float. You can compare this value with the version of the currently executing code to assure compatibility.
- Filename of wrapper module library.
- A wrapper information object containing information pertaining to this wrapper that was stored in the database manager catalog as a result of executing CREATE WRAPPER or ALTER WRAPPER DDL statements.

Only member functions should inspect or alter this information.

Required customization for all wrappers

The fenced generic wrapper subclass must implement:

- A constructor, which calls the corresponding fenced generic wrapper base class constructor.

  C++: Fenced_Generic_Wrapper

  Java: FencedGenericWrapper
- create server: a method to create an instance of your fenced generic server subclass.

  C++: Fenced_Generic_Server

  Java: FencedGenericServer

Additional customization

- If you need to store additional information in instances of your fenced generic wrapper subclass, override the default implementation of the initialize_my_wrapper function to extract this information from the wrapper information object supplied as a parameter. In C++, you cannot retain a pointer to this wrapper information object. If you choose to refer to the information in this form, use the data member that contains a copy of the wrapper information object. In Java, the wrapper can retain a reference to the WrapperInfo object but there is no copy function.
- If your Unfenced_Generic_Wrapper subclass points to any out-of-line storage you allocated, you must implement a destructor for your subclass which frees this storage.

Table 18. Virtual functions

Virtual function in C++	Virtual function in Java	Default provided?	Default behavior
initialize_my_wrapper	initializeMyWrapper	Yes	No-op
create_server	createServer	Yes	Error

Related tasks:

- "Server classes" on page 72
- "Nickname classes" on page 76
- "User classes" on page 80
- "Altering a wrapper" in the *Federated Systems Guide*

Related reference:

- "ALTER WRAPPER statement" in the *SQL Reference, Volume 2*
- "Wrapper classes for the Java API" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Wrapper class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Wrapper classes for the C++ API" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Server classes

The following sections describe the Unfenced_Generic_Server and fenced generic server classes.

Unfenced_Generic_Server class

Invoking the create server method on an instance of your unfenced wrapper subclass creates an instance of your unfenced server subclass. The federated server calls this method prior to the application's first use of the relevant server. If a DDL operation changes information pertaining to a server, the federated server will destroy and recreate the server object before its next use.

Unless otherwise customized, the Unfenced_Generic_Server base class implementation maintains the following information:

- Server name.
- A pointer to the appropriate wrapper object.
- A server information object containing information pertaining to the server that was stored in the federated server's system catalog as a result of executing DDL statements.

You can access some of this information directly as data members. Only data member functions can inspect or alter the rest of this information.

Required customization for all wrappers

The Unfenced_Generic_Server subclass must implement:

- A constructor for the subclass, which calls the corresponding Unfenced_Generic_Server constructor.
- create nickname: a method to create an instance of your unfenced nickname subclass.
- plan request: a method to analyze a query fragment contained in a Request object, and return one or more Reply objects each identifying a query fragment that can be executed by the data source and its corresponding execution cost.

Additional customization

- If you need to store additional information in instances of your Unfenced_Generic_Server subclass, override the default implementation of the initialize_my_server member function to extract this information from the server information object supplied as a parameter. In C++, you cannot retain a pointer to this server information object. If you chose to store the information in this form, use the data member that contains a copy of this information object. In Java the server can retain a reference to the ServerInfo object, and there is no copy function.
- If you define server options, server type, and server version, in addition to those already defined for the Unfenced_Generic_Server class, override the default implementations of the verify_my_register_server_info object and verify_my_alter_server_info object to verify the validity of the options and values supplied on the DDL. You do not need to verify the standard Unfenced_Generic_Server options; the Unfenced_Generic_Server implementation will do this. If you wish to alter an option value supplied on the DDL, or supply additional options, your implementation should supply the overriding/additional information via a "delta" server information object. In C++, before allocating a new "delta"object, check whether one already exists. If it does, do not allocate another. In Java, the "delta" wrapper information object is actually the return object of the verifyMyRegisterWrapperInfo and verifyMyAlterWrapperInfo methods. This object must be created by the wrapper.
- If the default Unfenced_Generic_User class is not sufficient for your wrapper, override the implementation of the create remote user method to create an instance of your Unfenced_Generic_User subclass.

- If the default cost model produces inaccurate estimates for cardinality or execution time, override the implementation of the create reply method to create an instance of a Reply subclass that you have customized to support your custom cost model.
- If the default selectivity estimator produces inaccurate values for predicates involving data from your data source, override the default implementation of the get selectivity method with a method that produces more accurate results.
- If your Unfenced_Generic_Server subclass points to any out-of-line storage you have allocated, you must implement a destructor for your subclass which frees this storage.

Table 19. Virtual functions

Virtual function in C++	Virtual function in Java	Default provided?	Default behavior
verify_my_register_server_info	verifyMyRegisterServerInfo	Yes	Verifies the options against the predefined list (according to the is_reserved_server_option() function in C++ and the isReserved() function in the CatalogOption class in Java). Any other options produce an error.
verify_my_alter_server_info	verifyMyAlterServerInfo	Yes	Verifies the options against the predefined list (according to the is_reserved_server_option() function in C++ and the isReserved() function in the CatalogOption class in Java). Any other options produce an error.
initialize_my_server	initializeMyServer	Yes	No-op
create_nickname	createNickname	Yes	Error
create_remote_user	createRemoteUser	Yes	Creates unfenced generic user
create_reply	createReply	Yes	Creates Reply supporting default cost model
get_selectivity	getSelectivity	Yes	Calculates selectivity with default model
get_type	getType	Yes	Gets type from information
get_version	getVersion	Yes	Gets version from information
plan_request	planRequest	No	Returns the appropriate reply objects

Fenced_Generic_Server class

Invoking the create server method on an instance of your fenced wrapper subclass creates an instance of your fenced server subclass. The federated server calls this method prior to the application's first use of the relevant server. If a DDL operation changes information pertaining to a server, the federated server will destroy and recreate the server object before its next use.

Unless otherwise customized, the fenced generic server base class implementation maintains the following information:

- Server name.
- A pointer to the appropriate wrapper object.
- A server information object containing information pertaining to this server that was stored in the federated server's system catalog as a result of executing DDL statements.
- A table of instantiated remote connection objects for this server (The current implementation limits the number of remote connections to one.)
- If a connection to the data source exists, a pointer to a fenced generic user object representing the user mapping for the connected user.

You can access some of this information directly as data members. Only data member functions can inspect or alter the rest of this information.

Required customization for all wrappers

The fenced generic server subclass must implement:

- A constructor for the subclass, which calls the corresponding fenced generic server constructor.
- create nickname: a method to create an instance of your fenced nickname subclass.

  C++: Fenced_Nickname

  Java: FencedNickname
- create remote connection: a method to create an instance of your remote connection subclass.

  C++: Remote_Connection

  Java: RemoteConnection

Additional customization

- If you need to store additional information in instances of the fenced generic server subclass, override the default implementation of the initialize_my_server function to extract this information from the server information object supplied as a parameter. In C++, you cannot retain a pointer to this server information object, instead use the data member that contains a copy of this object, should you choose to store the information in this form. In Java, the server can retain a reference to the ServerInfo object and there is no copy function
- If the default fenced generic user class is not sufficient for your wrapper, override the implementation of the create remote user method to create an instance of your fenced generic user subclass.

  C++: Fenced_Generic_User

  Java: FencedGenericRemoteUser
- If the fenced generic server subclass points to any out-of-line storage you have allocated, you must implement a destructor for your subclass which frees this storage.

Table 20. Virtual functions

Virtual function in C++	Virtual function in Java	Default provided?	Default behavior
initialize_my_server	initializeMyServer	Yes	No-op

Table 20. Virtual functions (continued)

Virtual function in C++	Virtual function in Java	Default provided?	Default behavior
create_remote_user	createRemoteUser	Yes	Creates the unfenced generic user class
create_remote_connection	createRemoteConnection	Yes	Error
create_nickname	createNickname	Yes	Error
get_type	getType	Yes	Gets type from information
get_version	getVersion	Yes	Gets version from information

Table 21. Public/protected member functions

Public/protected member functions in C++	Public/protected member functions in Java	Behavior
find_current_remote_user	findRemoteUser	Finds the remote user object for the current author id
find_connection	findConnection	Gets the connection object for a single connected remote user
find_nickname	findNickname	Gets the unfenced nickname object given a local schema name

Related tasks:
- "Wrapper classes" on page 69
- "Nickname classes" on page 76
- "User classes" on page 80

Related reference:
- "Server classes for the Java API" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Server classes for the C++ API" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Nickname classes

The following sections describe the unfenced generic nickname and fenced generic nickname classes.

Unfenced_Generic_Nickname class

Invoking the create nickname method on an instance of the fenced server subclass creates an instance of the unfenced nickname subclass. This method is usually called as a result of a call to the unfenced generic server method find_nickname (findNickname in Java). If a DDL operation changes information pertaining to a nickname, the federated server will destroy and recreate the nickname object before its next use.

Unless otherwise customized, the unfenced generic nickname base class implementation maintains the following information:
- The local name for the remote data set for which the nickname is being defined.

- The local schema for the remote data set for which the nickname is being defined.
- A pointer to the appropriate server object.
- Estimated values for four statistics that the default cost model uses:
  1. The cardinality of a nickname. This is defined as the number of rows contained in the nickname. The federated server stores the cardinality for an individual nickname in the system table SYSCAT.TABLES or SYSSTAT.TABLES (the “CARD” column in either table.) If cardinality is not available for a nickname, the cost model uses a default value of 1000 rows.
  2. The setup cost for a nickname. Setup cost represents the typical time, in milliseconds, that it takes a wrapper to get a query fragment ready to submit to the remote source. Setup begins when a wrapper receives the wrapper Execution Descriptor it produced during query planning, and ends when the wrapper is ready to submit the corresponding operation to the remote source. Setup cost should only include work that the wrapper does not need to repeat if the wrapper is asked to perform the same query fragment again, perhaps with a different parameter value. For example, if a wrapper submits query fragment to a remote source in the form of a URL, setup cost includes the time required to generate that URL from the information stored by the wrapper in the Execution Descriptor. The federated server stores this statistic in the SETUP_COST nickname option. If that option is not present for a nickname, then the cost model uses a value of 2000 milliseconds.
  3. The submission cost for a nickname. Submission cost represents the typical time, in milliseconds, that it takes a wrapper to submit a query fragment to the remote source. Submission begins at the end of setup, as defined above, and ends when the wrapper is ready to request the first row or block of result data from the source. Submission cost should only include work that the wrapper must repeat each time a given query fragment is submitted. For example, if a new HTTP connection is required for each interaction with the remote source, submission cost should include the time necessary to create this connection. The federated server stores this statistic in the SUBMISSION_COST nickname option. If that option is not present for a nickname, then the cost model uses a value of 25 milliseconds.
  4. The advance cost for a nickname. This is the typical time, in milliseconds, that it takes to fetch a single row for the nickname. It is exclusive of any time necessary to start a query. The federated server stores this statistic in the ADVANCE_COST nickname option. If that option is not present for a nickname, then the cost model uses a value of 50 milliseconds. If the data source returns data in blocks, rather than rows, calculate the advance cost by dividing the typical cost of fetching a block by the typical number of rows per block.

Required customization for all wrappers

The Unfenced_Generic_Nickname subclass must implement:

- A constructor for the subclass, which should just call the corresponding Unfenced_Generic_Nickname constructor.

Additional customization

- If you need to store additional information in instances of the Unfenced_Generic_Nickname subclass, override the default implementation of initialize_my_nickname function to extract this information from the nickname information object supplied as a parameter. In C++, you cannot retain a pointer to this nickname information object. In Java, you can retain a reference to the nickname information object.

- If you define nickname options or column options, override the default implementations of verify_my_register_nickname_info and verify_my_alter_nickname_info (verifyMyRegisterNicknameInfo and verifyMyAlterNicknameInfo in Java) to verify the validity of the options and values supplied on the DDL. If you wish to alter an option value supplied on the DDL or supply additional options, the implementation should supply the overriding/additional information via a “delta” nickname information object. Before allocating a new “delta” object, check whether one already exists. If so, use it and do not allocate another. In Java, the “delta” object is actually a return object and must be created by the wrapper.
- If you allocated any out-of-line storage pointed to by the Unfenced_Generic_Nickname subclass, you must implement a destructor for the subclass which frees this storage.

Table 22. Virtual functions

Virtual function in C++	Virtual function in Java	Default behavior
initialize_my_nickname	initializeMyNickname	No-op
verify_my_register_nickname_info	verifyMyRegisterNicknameInfo	Verifies the options against the predefined list (according to the is_reserved_nickname_option() function in C++ and the CatalogOption.isReserved() method in Java). Any other options produce an error.
verify_my_alter_nickname_info	verifyMyAlterNicknameInfo	Verifies the options against the predefined list (according to the is_reserved_nickname_option() function in C++ and the CatalogOption.isReserved() method in Java). Any other options produce an error.

Fenced_Generic_Nickname class

Invoking the create nickname method on an instance of the fenced server subclass creates an instance of the fenced nickname subclass. This method is usually called as a result of executing a CREATE NICKNAME statement. If a DDL operation changes information pertaining to a nickname, the federated server will destroy and recreate the nickname object before its next use.

Unless otherwise customized, the Fenced_Generic_Nickname base class implementation maintains the following information:

- The local name for the remote data set for which the nickname is being defined.
- The local schema for the remote data set for which the nickname is being defined.
- A pointer to the appropriate server object.

Required customization for all wrappers

The Fenced_Generic_Nickname subclass must implement:

- A constructor for the subclass, which calls the corresponding Fenced_Generic_Nickname constructor.

Additional customization

- If you need to store additional information in instances of the Fenced_Generic_Nickname subclass, override the default implementation of the initialize_my_nickname function to extract this information from the nickname information object supplied as a parameter. In C++, you cannot retain a pointer to this nickname information object. In Java, you can retain a reference to the nickname information object.
- If you define nickname options, override the default implementation of the verify_my_register_nickname_info function to verify that the validity of the options and values supplied on the DDL. If you wish to alter an option value supplied on the DDL, supply additional options, or supply information and options obtained from the data source rather than from DDL, your implementation should supply the overriding or additional information using a "delta" nickname information object. Before allocating a new "delta" object, check whether one already exists. If so, use it and do not allocate another. In Java, the "delta" object is actually a return object and must be created by the wrapper.
- If you allocated any out-of-line storage pointed to by the Unfenced_Generic_Nickname subclass, you must implement a destructor for the subclass which frees this storage.

Table 23. Virtual Functions

Virtual function in C++	Virtual function in Java	Default behavior
initialize_my_nickname	initializeMyNickname	No-op
verify_my_register_nickname_info	verifyMyRegisterNicknameInfo	Verifies the options against the predefined list (according to the is_reserved_nickname_option() in C++ and the CatalogOption.isReserved() method in Java). Any other options produce an error.

Table 24. Public/protected member functions

Public/protected member function in C++	Public/protected member function in Java	Behavior
get_card	getCard	Accessor
get_setup_cost	n/a	Accessor
get_advance_cost	n/a	Accessor
get_submission_cost	n/a	Accessor

Related tasks:

- "Wrapper classes" on page 69
- "Server classes" on page 72
- "User classes" on page 80

Related reference:

- "Nickname classes for the Java API" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Nickname class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Nickname classes for the C++ API" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

User classes

The following sections describe the Unfenced_Generic_User and fenced generic user classes.

Unfenced_Generic_User class

Invoking the create remote user method on an instance of your unfenced server subclass creates an instance of your unfenced user subclass. The federated server calls this method when processing CREATE USER MAPPING or ALTER USER MAPPING statements. If a DDL operation changes information pertaining to a remote user, the federated server will destroy and recreate the remote user object before its next use.

Unless otherwise customized, the Unfenced_Generic_User base class implementation maintains the following information:

- The local authid of the user.
- A user information object containing information pertaining to this (server, user) pair that was stored in the federated server's system catalog as a result of executing DDL statements.
- A pointer to the appropriate server object.

Required customization for all wrappers

If your data source does not require authentication information, customization of the unfenced generic user base class is not necessary.

If you create an Unfenced_Generic_User subclass, it must implement:

- A constructor for your subclass, which should just call the corresponding Unfenced_Generic_User constructor.

Additional customization

- If you need to store additional information in instances of your Unfenced_Generic_User subclass, override the default implementation of the initialize_my_user function to extract this information from the user information object supplied as a parameter. In C++, you cannot retain a pointer to this user information object, instead use the data member that contains a copy of this object, should you choose to store the information in this form. In Java, you can retain a reference to the UserInfo object but there is no copy functionality.
- If you define user options, override the default implementations of the verify_my_register_user_ info and the verify_my_alter_user_info functions to verify the validity of the options and values supplied on the DDL. If you wish to alter an option value supplied on the DDL, or supply additional options, your implementation should supply the overriding/additional information via a "delta" user information object. Before allocating a new "delta" object, check whether one already exists. If so, use it and do not allocate another. In Java, the "delta" object is actually the object returned by the verify methods and needs to be created by the wrapper.
- If your Unfenced_Generic_User subclass points to any out-of-line storage you have allocated, you must implement a destructor for your subclass which frees this storage.

Table 25. Virtual Functions

Virtual function in C++	Virtual function in Java	Default provided?	Default behavior
verify_my_register_user_info	verifyMyRegisterUserInfo	Yes	Verifies that only DB2 UDB options have been specified.
initialize_my_user	initializeMyUser	Yes	No-op
verify_my_alter_user_info	verifyMyAlterUserInfo	Yes	Verifies that only DB2 UDB options have been specified.

Table 26. Public/protected member functions

Public/protected member function in C++	Public/protected member function in Java	Behavior
get_local_name	getLocalName	Accessor

Fenced_Generic_User class

Invoking the create remote user method on an instance of your fenced server subclass creates an instance of your fenced user subclass. This method is called by the federated server prior to the application's first request to connect to the server using the remote connection object. If a DDL operation changes information pertaining to a remote user, the federated server will destroy and recreate the remote user object before its next use.

Unless otherwise customized, the fenced generic user base class implementation maintains the following information:

- The local authid of the user.
- A user information object containing information pertaining to this server/user pair that was stored in the federated server's system catalog as a result of executing DDL statements.
- A pointer to the appropriate server object.

Required customization for all wrappers

If your data source does not require authentication information, customization of the Fenced_Generic_User base class is not necessary.

If you create a Fenced_Generic_User subclass, it must implement:

- A constructor for your subclass, which should just call the corresponding Fenced_Generic_User constructor.

Additional customization

- If you need to store additional information in instances of your fenced generic user subclass, override the default implementation of the initialize_my_user function to extract this information from the user information object supplied as a parameter. In C++, you cannot retain a pointer to this user information object, instead use the data member that contains a copy of this object, should you choose to store the information in this form. In Java, you can retain a reference to the UserInfo object, but there is no "copy" functionality.
- If your Fenced_Generic_User subclass points to any out-of-line storage you have allocated, you must implement a destructor for your subclass which frees this storage.

Table 27. Virtual functions

Virtual function in C++	Virtual function in Java	Default provided?	Default behavior
initialize_my_user	initializeMyUser	Yes	No-op

Table 28. Public/Protected member functions

Public/protected member function in C++	Public/protected member function in Java	Behavior
get_local_name	getLocalName	Accessor

Related tasks:
- "Wrapper classes" on page 69
- "Server classes" on page 72
- "Nickname classes" on page 76

Related reference:
- "User classes for the Java API" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "User classes for the C++ API" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Request class

This class is used during query planning as part of the Request-Reply-Compensate protocol. The federated server optimizer generates an instance of this class to describe each query fragment that the data source might be asked to evaluate.

Methods

Table 29. Methods

Name in C++	Name in Java	Description
get_number_of_quantifiers	getNumberOfQuantifiers	Retrieves the number of elements in the FROM clause.
get_number_of_predicates	getNumberOfPredicates	Retrieves the number of elements in the WHERE clause.
get_number_of_head_exp	getNumberOfHeadExp	Retrieves the number of elements in the SELECT clause.
get_quantifier_handle	getQuantifier.getHandle	Retrieves a handle for the element at a specified position in the FROM clause.
get_predicate_handle	getPredicate.getHandle	Retrieves a handle for the element at a specified position in the WHERE clause.
get_head_exp_handle	getHeadExp.getHandle	Retrieves a handle for the element at a specified position in the SELECT clause.
get_nickname	getNickname	Retrieves a Nickname object for a quantifier designated by a handle in the FROM clause.
get_head_exp	getHeadExp	Retrieves a Request Exp object for a head expression designated by a handle in the SELECT clause.

Table 29. Methods (continued)

Name in C++	Name in Java	Description
get_predicate	getPredicate	Retrieves a Request Exp object for a predicate designated by a handle in the WHERE clause.
get_distinct	getDistinct	Tests the DISTINCT indicator.

Related reference:

- "Request class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Request classes for the C++ API" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Reply class

A wrapper generates an instance of this class to represent a portion of the query fragment in a request that the data source can run. A wrapper can generate several replies for a single Request.

The Reply class adds methods and data members to:

- populate the reply by adding entries to the classes: add_to_CCCC
- store the execution descriptor pointer and size
- chain the replies when the wrapper returns more than one plan for the same request
- store order information: interesting orderings of the returned data that the optimizer might use to construct more optimal plan

Advanced customization

You can override the default implementation of the costing methods with ones that more precisely describe the source's execution model. Overriding the default costing is dependent on the default selectivity estimation for the predicates in the query fragment. Calling the Unfenced_Generic_Server::get_selectivity method (UnfencedGenericServer.getSelectivity method in Java) obtains selectivity estimates that you can also override.

Methods

Table 30. Methods

Name in C++	Name in Java	Default provided?	Description
add_head_exp	addHeadExp	Yes	Adds a head expression handle at the next free position.
add_predicate	addPredicate	Yes	Adds a predicate handle at the next free position.
add_quantifier	addQuantifier	Yes	Adds a quantifier handle at the next free position
add_order_entry	addOrderEntry	Yes	Adds an order entry. Order entry is composed of a direction (ASC or DSC) and integer index of the head expression ordered.

Table 30. Methods (continued)

Name in C++	Name in Java	Default provided?	Description
get_parameter_order	n/a	Yes	Returns an order of the parameters for replies that represent query fragments requiring parameters. It performs a preorder traversal of all the expressions in the WHERE and SELECT clauses and produces an order. Returns a list of parameter handles.
cardinality	cardinality	Yes	Returns the cardinality of the result returned by executing the query fragment represented by the reply. The default version of cardinality() returns the product of the cardinalities of all of the nicknames multiplied by the selectivity of the predicates.
total_cost	totalCost	Yes	Total execution cost, in milliseconds, needed to execute the query fragment represented by the reply. The default version calls all_costs and returns the total cost value.
re_exec_cost	reExecCost	Yes	Time needed to re-execute the query fragment represented by the reply. The default calls all_costs() and returns the re-execute cost value. **Note:** It is better if the developer overrides all_costs() than override this routine.
first_tuple_cost	firstTupleCost	Yes	Time needed to obtain the first result tuple for the query fragment represented by the reply. The default calls all_costs() and returns the first-tuple cost value **Note:** It is better if the developer overrides all_costs() than override this routine.
set_next_reply	setNextReply	Yes	Chain the replies when the wrapper returns more than one reply for a request.

Table 30. Methods (continued)

Name in C++	Name in Java	Default provided?	Description
all_costs	n/a	No; you must implement	Calculates three cost values, total cost, first tuple cost and re-execute cost. **first tuple cost** = average-setup-cost + average-submission-cost + average-advance-cost. **re-execute cost** = average-submission-cost + (average-advance-cost * estimated-cardinality) **total cost** = average-setup-cost + average-submission-cost + (average-advance-cost * estimated-cardinality) where: **average-setup-cost** The average of the setup costs for all nicknames in the query fragment. **average-submission-cost** The average of the submission costs for all nicknames in the query fragment. **average-advance-cost** The average of the advance costs for all nicknames in the query fragment. **estimated-cardinality** The estimated cardinality (returned by the cardinality() method) for the query fragment.
get_number_of_quantifiers	getNumberOfQuantifiers	Yes	Retrieves the number of elements in the FROM clause.
get_number_of_predicates	getNumberOfPredicates	No; you must implement	Retrieves the number of elements in the WHERE clause.
get_number_of_head_exp	getNumberOfHeadExp	No; you must implement	Retrieves the number of elements in the SELECT clause.
get_quantifier_handle	getQuantifier.getHandle	No; you must implement	Retrieves a handle at a position in the FROM clause.
get_predicate_handle	getPredicate.getHandle	No; you must implement	Retrieves a handle at a position in the WHERE clause.
get_head_exp_handle	getHeadExp.getHandle	No; you must implement	Retrieves a handle at a position in the SELECT.
get_nickname	getNickname	No; you must implement	Retrieves a class Nickname object for quantifier at a given handle in the FROM clause.
get_head_exp	getHeadExp	No; you must implement	Retrieves a class Request_Exp object for a head expression at a given handle in the SELECT clause.

Table 30. Methods (continued)

Name in C++	Name in Java	Default provided?	Description
get_predicate	getPredicate	No	Retrieves a class Request_Exp object for a predicate at a given handle in the WHERE clause.
set_distinct	setDistinct	No; you must implement	Retrieves the DISTINCT indicator (SELECT DISTINCT clause).
get_distinct	getDistinct	No; you must implement	Tests the DISTINCT indicator.
get_exec_desc	getExecDistinct	No; you must implement	Retrieves a pointer and length for the execution descriptor for this query.
set_exec_desc	setExecDesc	No; you must implement	Stores an execution descriptor in the Reply.

Related reference:

- "Reply class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Reply class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Predicate list class

Instances of the predicate list class are input to the selectivity estimation method, Unfenced_Generic_Server::getSelectivity() (UnfencedGenericServer.getSelectivity in Java). The two predicate lists are:

- a set of predicates, P, for which selectivity is solicited.
- a set of predicates, AP, that have already been applied.

The result selectivity is the *conditional* selectivity of P given AP, that is the selectivity of the predicates in set P given that the predicates in set AP have already been applied:

```
selectivity(P/AP)
```

If unconditional selectivity is required, AP can be NULL. The list of predicates are similar to the list of predicates in the Request and are manipulated with similar methods.

Methods

Table 31. Methods

Name in C++	Name in Java	Description
create_empty_predicate_list	n/a	Creates an empty predicate list, of the same size as the predicate list in a request. The resulting list is empty.
create_and_copy_predicate_list	n/a	Creates an empty predicate list and copies the predicate list from the reply.
operator new	n/a	Allocates the predicate list on the QG heap.

Table 31. Methods (continued)

Name in C++	Name in Java	Description
destructor	n/a	Predicate_List objects are created by the wrapper and must be destroyed by the wrapper.
get_number_of_predicates	getNumberOfPredicates	Retrieves the length of the predicate list.
get_predicate	getPredicate	Returns a Request_Exp object describing the predicate at a handle.
get_predicate_handle	getPredicateHandle	Returns the handle of a predicate at a position.
get_number_of_applied_predicates	getNumberOfAppliedPredicates	Returns the length of the applied predicate list.
get_applied_predicate	getAppliedPredicate	Returns a Request_Exp object describing the applied predicate at a handle.
get_applied_predicate_handle	getAppliedPredicateHandle	Returns the handle of an applied predicate at a position.
add_predicate	addPredicate	Adds predicate handle to the next free position in the predicate list.
add_applied_predicate	addAppliedPredicate	Adds applied predicate handle to the next free position in the applied predicate list.

Related reference:

- "PredicateList class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Predicate_List class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Request expression class

This class defines the interface for exploring the request expressions. An expression can be a head expression or a predicate. All these expressions are represented as parsed operator trees. Since expressions are recursively defined, each tree node itself is an expression. The interior nodes (subexpressions) of a tree represent operators while the leaves represent constants, columns, or parameters. Depending on the type of the expressions, different information is available to the wrapper writer. For each kind of expression, different information is available to the wrapper. In addition to a method that returns the kind, all expressions have methods that return the data type of the expression, its parent expression (NULL for the root expression) and its sibling expression (next child to the right if it exists, otherwise NULL). The individual kinds of expressions have methods that return additional information, as described in the following table:

Table 32. Additional information for each type of expression

Type of expression	Information
Operators	• number of children (operands) • first operand • token (as parsed from the query) • signature - resolved function/operator name including the operand data types
Column	• column name • quantifier to which the column belongs
Constant	Value buffer, length, and type
Parameter	Parameter handle used to identify the position of the parameter in the parameter array used to pass the parameters from the engine to the wrapper.

Methods

Table 33. Request expression class methods in C++

Name in C++	Description
get_kind	Returns the kind of the expression (Request_Exp::oper, Request_Exp::column, Request_Exp::unbound, Request_Exp::constant). The last argument to the function is needed only when called from the optimizer.
get_data_type	Returns Request_Exp_Type instance describing the result data type of the expression.
get_parent	Returns the parent node of the expressions.
get_next_child	Returns the sibling node.
get_column_name	Returns a string with the column name.
get_column_quantifier_handle	Returns the quantifier to which this column belongs.
get_value	Returns a Request_Constant instance representing the value and the type of a constant in the expression.
get_parameter_handle	Returns an integer uniquely identifying a parameter.
get_number_of_children	Returns the number of operands of an operator expression.
get_first_child	Returns the first operand.
get_token	Returns the token value as a string.
get_signature	Returns the signature of an operator.

Table 34. Request expression class methods in Java

Name in Java	Description
getKind	Returns the kind of the expression (RequestExp.OPERATOR, RequestExp.CONSTANT, RequestExp.UNBOUND, RequestExp.COLUMN).
getDataType	Returns RequestExpType instance describing the result data type of the expression.
getParent	Returns the parent node of the expressions.
getNextChild	Returns the sibling node.
getColumnName	Returns a string with the column name.

Table 34. Request expression class methods in Java (continued)

Name in Java	Description
getQuantifier	Returns the quantifier to which this column belongs.
getValue	Returns a RequestConstant instance representing the value of a constant in the expression.
getNumberOfChildren	Returns the number of operands of an operator expression.
getFirstChild	Returns the first operand.
getToken	Returns the token value as a string.
getSignature	Returns the signature of an operator.
getHandle	Returns the handle of the expression.

Related reference:

- "RequestExp class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Request_Exp class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Request constant class

You use this class to describe constants in a query expression. The Request_Exp::get_value() (RequestExp getValue in Java) function returns an instance of the request constant class when applied to a node of Request_Exp::constant kind (RequestExp.CONSTANT in Java).

Methods

Table 35. Methods

Name in C++	Name in Java	Description
get_actual_length	getActualLength	Gets the length of data in the buffer.
get_data	getData	Gets a pointer to the data buffer.
get_data_type	getDataType	Returns the DB2 UDB type ID of the column
is_data_null	isDataNull	Semantic or friendly arithmetic null?
get_precision	getPrecision	Returns the precision of a numeric constant value.
get_scale	getScale	Returns the scale of a numeric constant value.
get_codepage	getCodepage	Returns the codepage of a character value.

Related reference:

- "RequestConstant class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Request_Constant class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Request expression type class

Instances of this class describe the data type of a query expression. Each query expression, including the subexpressions, is typed. The function Request_Exp::get_data_type() (RequestExpType getDataType in Java) returns a request expression type instance.

Methods

Table 36.

Name in C++	Name in Java	Description
get_for_bit_data	getForBitData	Does column contain binary data? Valid values are Y and N where Y is Yes and N is No. The Java method getForBitData returns a boolean value (not Y and N).
get_null_indicator	getNullIndicator	Specifies which column is the null indicator
get_data_type	getDataType	In C++, SQL type of column (from sql.h). In Java, the type ID constants are defined in the Data class.
get_maximum_length	getMaximumLength	Max length of column (except for SQL_TYP_DECIMAL)
get_precision	getPrecision	Precision (SQL_TYP_DECIMAL only)
get_scale	getScale	Scale (SQL_TYPE_DECIMAL only)
get_codepage	getCodepage	Codepage for column
get_name	getName	Column name (if any)
get_name_length	getNameLength	Length of column name

Related reference:
- “RequestExpType class (Java)” in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- “Request_Exp_Type class (C++)” in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Remote connection class

To create a remote connection instance, invoke the method create_remote_connection (createRemoteConnection in Java) on an instance of the appropriate fenced server subclass. The federated server calls this method prior to processing the first remote operation at the relevant server. The federated server destroys the remote connection instance after the application has executed a prescribed number of transactions without making use of the server.

Unless otherwise customized, the remote connection base class implementation maintains the following information:
- The status (connected or disconnected) of your connection to your data source.
- A pointer to the appropriate server object.
- A pointer to the appropriate user object.
- A list of remote operation objects that are instantiated for this connection.

- The code page to use for this connection. This is the code page for the client application connected to the federated server.
- The connection kind: no-phase, one-phase, two-phase (For Version 8.2, two-phase transactions are not supported).

Required customization for all wrappers

The remote connection subclass must implement a constructor for the subclass, which calls the corresponding remote connection constructor.

The remote connection class might also need to implement other methods, depending on the needs of the wrapper. The following list describes the constructors and their use:

- connect: a method to establish a connection to your data source. You must implement this method even if your data source does not require a connection.
- disconnect: a method to close an established connection to your data source. You must implement this method even if your data source does not require a connection.

  A call to the mark_disconnected method (markDisconnected method in Java) is not necessary after you establish or close the connection.
- Create remote query method: creates an instance of the remote query subclass.

  C++: create_remote_query

  Java: createRemoteQuery
- commit: a method to commit the current transaction at your data source. If the connection is being used for a passthru session and the data source supports transactions, your implementation should commit the current transaction at the data source. All non-passthru transactions are read-only in version 8.1, but your data source requires an end-of-transaction notification to free resources, drop read locks, and so on. You must implement this method even if your data source does not support transactions.
- rollback: a method to roll back the current transaction at the data source. If the connection is being used for a passthru session and the data source supports transactions, your implementation should roll back the current transaction at the data source. All non-passthru transactions are read-only in version 8.1, but your data source requires an end-of-transaction notification to free resources, drop read locks, and so on. You must implement this method even if your data source does not support transactions.

In addition, if any part of your implementation detects that your connection to the data source is lost, it should call the mark disconnected method and return an error. The mark disconnection method informs the database manager that the connection is lost and starts the appropriate cleanup.

Additional customization

- If you need to store information about a connection to your data source, such as a connection handle, add appropriate data members to your subclass definition and initialize them in the constructor.
- If you allocated any out-of-line storage pointed to by your remote connection subclass, you must implement a destructor for your subclass which frees this storage. Your destructor does not need to close the connection to the data source; if a connection to the data source might exist, the federated server will always call the disconnect method before deleting a remote connection object.

- If your wrapper supports passthru, you must override the default implementation of the create remote passthru method. Your implementation should create an instance of your remote passthru subclass.

Table 37. Virtual Functions

Virtual function in C++	Virtual function in Java	Default provided?	Default behavior
connect	connect	No; you must implement	
disconnect	disconnect	No; you must implement	
create_remote_query	createRemoteQuery	Yes	Error
create_remote_passthru	createRemotePassthru	Yes	Error
commit	commit	No; you must implement	
rollback	rollback	No; you must implement	

Table 38. Public/protected Member Functions

Public/protected member function in C++	Pub./Prot. member function in Java	Behavior
is_connected	isConnected	Connected to data source?
mark_connected	markConnected	Connected to data source
mark_disconnected	markDisconnected	Not connected to data source.
get_kind	getKind	Accessor
get_codepage	getCodepage	Accessor

Related reference:

- "Remote_Connection class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*
- "RemoteConnection class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*

Remote query class

Invoking the method create remote query class on an instance of the appropriate remote connection subclass creates an instance of your remote query subclass. The federated server destroys the remote query object when the application that submitted the query closes its cursor over the query's result set.

For some data sources, a wrapper will not be able to control the order in which the data source returns data values. The wrapper might get non-LOB data followed by LOB data, followed by more non-LOB data, and so forth. The wrapper has no choice but to handle data in the order the data is received from the data source.

You must implement the constructor for the subclass, which calls the corresponding remote query constructor. The output runtime data list is created by the remote operation base class constructor. The remote query subclass must implement the following methods. The following explanations use the C++ names; if you are developing a wrapper in Java substitute the corresponding Java method name.

fetch() The fetch() method is responsible for retrieving non-LOB data into the federated server buffers that are identified by the runtime data list.

fetch_lob()
The fetch_lob() method is responsible for retrieving LOB data, one chunk at a time, into a special LOB buffer that the federated server provides. The fetch_lob() method might require several calls to process the entire LOB, because the size of the buffer limits the amount that is transferred on each call.

fetch() and fetch_lob()
The fetch() and fetch_lob() methods operate in a coordinated fashion to handle LOB and non-LOB columns in the order in which they arrive from the remote source. The remote query object provides shared state, so the wrapper can keep track of where it is in the data stream. Before returning from either fetch() or fetch_lob(), the wrapper indicates to the gateway whether the next call from the federated server should be to fetch() or fetch_lob().

open_input_lob()
The open_input_lob() method allows the wrapper to initialize a remote query that contains input LOB parameters. You must implement this method in the wrapper-specific remote query class if the wrapper supports input LOB parameters. DB2 Information Integrator calls this method if it finds input LOB host variables. The wrapper must return the index of the host variable that receives the next LOB fragment and call set_row_status() to indicate that there are input LOB parameters.

While the wrapper indicates that there are more LOB input parameters to process, DB2 Information Integrator calls this method with another LOB fragment. The size, in bytes, of the current LOB fragment is passed to the wrapper, and the wrapper must use that information to advance to the next input variable or to signal that the entire input values have been read.

reopen_input_lob()
The reopen_input_lob() method resets a previously opened result stream and prepares the data source to return more result sets, possibly based on different parameter bindings for queries with input LOB parameters. You must implement this method in the wrapper-specific remote query class if the wrapper supports input LOB parameters. This method is not called unless the query was previously closed with an end-of-query status (the close method). DB2 Information Integrator calls the reopen_input_lob() method if input LOB host variables are found. The wrapper must return the index of the host variable that receives the next LOB fragment and call set_row_status() to indicate that there are input LOB parameters.

While the wrapper indicates that there are more LOB input parameters to process, DB2 Information Integrator calls this method with another LOB fragment. The size, in bytes, of the LOB input value and the current LOB fragment are passed to the wrapper, and the wrapper must use these values to advance to the next input variable or to signal that the entire input values have been read.

set_row_status()
The get_row_status() method sets the current row status. This method is already implemented and cannot be overloaded by the wrapper.

get_row_status()
The set_row_status() method retrieves the current row status. This method is already implemented and cannot be overloaded by the wrapper.

You have considerable discretion in how you want to distribute function among the methods here.

Table 39. Functions

Virtual function in C++	Virtual function in Java	Default provided?	Default behavior
fetch	fetch	Yes	Error
fetch_lob	fetchLob	Yes	Error
open	open	Yes	Error
open_input_lob	openInputLob	Yes	Error
reopen	reopen	Yes	Error
reopen_input_lob	reopenInputLob	Yes	Error
close	close	Yes	Error
set_row_status	setRowStatus	Yes	Error
get_row_status	getRowStatus	Yes	Error

Related reference:

- "RemoteQuery class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Remote_Query class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Runtime data classes

The following sections describe the runtime data and runtime data list classes.

Table 40. Runtime data list classes in C++ and Java

C++	Java
Runtime_Data	RuntimeData
Runtime_Data_List	RuntimeDataList

Runtime data class

The federated server creates an instance of the runtime data class to represent each buffer the system uses to transfer column values between the server and a wrapper. The federated server creates a buffer for each query parameter for which it transfers a value to the wrapper during execution. The federated server also creates a buffer for each column of the result row returned by the wrapper (for example, for each head expression in the query fragment).

The following state is maintained:

- Column number
- A pointer to a Runtime_Data_Desc object (RuntimeDataDesc in Java) describing the value.
- A pointer to a data buffer
- The length of the data in the buffer

- For columns representing parameter values, a flag indicating whether or not the parameter value is invariant. The value of an invariant parameter will not change unless the wrapper is notified via the “action” parameter of the “reopen” method.

The federated server supplies the column number, data description and data buffer. If the runtime data object is being used to pass a parameter from the federated server to a wrapper, the data length and buffer contents will be supplied by the federated server. The value in the buffer will have the SQL type specified by the attached description. The wrapper must convert the value to whatever type the data source expects. If the runtime data object is being used to return results from the wrapper to the federated server, the wrapper supplies the data length and buffer contents. The maximum (allocated) length of the buffer and the expected data type can be obtained from the attached description. The wrapper must convert the value obtained from the data source to the SQL type specified by the description.

Table 41. Public member functions

Public member function in C++	Public member function in Java	Behavior
check_friendly_div_by_0	checkFriendlyDivBy0	Division by zero?
check_friendly_exception	checkFriendlyException	Arithmetic exception?
get_actual_length	getActualLength	Get length of data in buffer
get_data	getData	Get pointer to data buffer
get_data_index	getDataIndex	Get column number
get_invariant	getInvariant	Is input value invariant?
is_data_null	isDataNull	Semantic or “friendly arithmetic” null?
is_semantic_null	isSemanticNull	Semantic null?
set_actual_length	setActualLength	Set length of data in buffer
set_data	setXX	Copy data to buffer
set_data_null	setDataNull	Set null indicator
set_friendly_div_by_0	setFriendlyDivBy0	Indicate zero-divide exception, set null first!
set_friendly_exception	setFriendlyException	Indicate arithmetic exception, set null first!

Runtime data list class

The base class constructor creates instances of this class for the remote query. Separate lists are created for output rows and input parameters.

Table 42. Public member functions

Public member function in C++	Public member function in Java	Behavior
get_number_of_values	getNumberOfValues	Get number of columns
get_ith_value	getValue	Get pointer to Runtime_Data for i-th column
operator[]	n/a	Get pointer to Runtime_Data for i-th column

Related reference:

- "Runtime_Data class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*
- "Runtime_Data_List class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*
- "RuntimeDataList class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "RuntimeData class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*

Runtime data description classes

The following sections describe the runtime data description and runtime data description list classes.

Table 43. Runtime data description list classes in C++ and Java

C++	Java
Runtime_Data_Desc	RuntimeDataDesc
Runtime_Data_Desc_List	RuntimeDataDescList

Runtime data description class

The federated server creates an instance of this class to represent each buffer the system uses to transfer column values to a wrapper. The federated server creates a buffer for each query parameter for which it transfers a value to the wrapper during execution. The federated server also creates a buffer for each column of the result row returned by the wrapper (for example, for each head expression in the query fragment). For both remote query and remote passthru objects, the federated server will create a runtime data description for each runtime data object in the input data list representing parameter values. The federated server supplies the column descriptions. For remote query objects, the federated server also creates a runtime data description for each runtime data object in the output data list, representing values in result rows. The federated server supplies these column descriptions as well.

Table 44. Public member functions

Public member function in C++	Public member function in Java	Behavior
get_for_bit_data	getForBitData	Does column contain binary data?
get_null_indicator	getNullIndicator	Can column be null?
get_data_type	getDataType	SQL type of column (from sql.h)
get_maximum_length	getMaximumLength	Max length of column (except for SQL_TYP_DECIMAL)
get_precision	getPrecision	Precision (SQL_TYP_DECIMAL only)
get_scale	getScale	Scale (SQL_TYPE_DECIMAL only)
get_codepage	getCodepage	Codepage for column
get_name	getName	Column name (if any)
get_name_length	getNameLength	Length of column name

Runtime data description list class

Table 45. Public member function

Public member function in C++	Public member function in Java	Behavior
get_number_of_values	getNumberOfValues	Get number of columns
get_ith_value	getValue	Get pointer to Runtime_Data_Desc for i-th column
set_ith_value	setValue	Set pointer to Runtime_Data_Desc for i-th column
operator[]	n/a	Get pointer to Runtime_Data_Desc for i-th column

Related reference:

- "Runtime_Data_Desc class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*
- "Runtime_Data_Desc_List class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*
- "RuntimeDataDesc class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "RuntimeDataDescList class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*

Remote passthru class

Invoking the method create remote passthru class on an instance of the appropriate remote connection subclass creates an instance of your remote passthru subclass. If your data source does not support passthru, do not implement a remote passthru subclass and do not override the default implementation of this method.

Required customization for all wrappers

Your remote passthru subclass must implement:

- A constructor for your subclass, which should call the corresponding remote passthru constructor.

  C++: Remote_Passthru

  Java: RemotePassthru
- prepare: a method that allows the passthru string to be submitted to the data source, to determine the number and type of the columns that will make up each row of the result.
- describe: a method that populates a runtime data description list that describes the number and type of the columns that make up each row of the result.
- open: a method that allows the data source to prepare to return the first result row for the query.
- fetch: a method that copies a single result row into the output runtime data list.
- close: a method that allows the data source to clean up after executing a query.

As in the case of remote query, the wrapper-writer has considerable discretion in the distribution of function among the methods listed previously. For example, population of the runtime data description list could occur in either the prepare or describe methods, and submission of the passthru string to the data source could occur in the open or fetch methods.

Additional customization

If your data source supports passthru strings that return a result code but no rows, override the default implementation of the execute method. Your implementation should submit the string to the data source and return the result code. If the client application submits a passthru string via the EXECUTE or EXECUTE IMMEDIATE language construct, the execute method will be called and should report an error if the passthru string does not represent an appropriate operation (that is if it represents an operation that returns rows)

Table 46. Virtual Functions

Virtual Function in C++ and Java	Default?	Default behavior
prepare	Yes	Error
describe	Yes	Error
execute	Yes	Error
open	Yes	Error
close	Yes	Error

Related reference:

- “RemotePassthru class (Java)” in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- “Remote_Passthru class (C++)” in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Wrapper utilities class

This class should not be instantiated. This class exists only to group together a collection of static methods that give wrappers access to miscellaneous services that are provided by the federated server.

Static methods of the wrapper utilities class provide the following services:

- Memory allocation and deallocation. Your implementation must use these methods to allocate and deallocate memory, except when using “new” to instantiate or “delete” to destroy a C++ object that is derived from the Sqlqg_Base_Class. If a class is not a descendant of Sqlqg_Base_Class, or you want to override the Sqlqg_Base_Class implementation of “new” or “delete”, you must provide implementations of these operators that use these methods to allocate and deallocate memory.
- Error reporting. It is the responsibility of the wrapper to map an error returned by a data source to the most appropriate DB2 UDB error or SQL code. An error message string exists for every SQL code and is frequently parameterized by tokens that are specific to the circumstances under which the error occurred. For example, the error message string for reporting a missing option is parameterized by tokens that contain the kind of option (wrapper, server, and so forth), the entity name (wrapper name, server name, and so forth), and the name

of the missing option. The wrapper should provide values for these tokens when reporting an error. To find the most appropriate SQL code and the content of the associated tokens, see the *Message Reference*.

After you identify the most appropriate SQL code, find a corresponding symbolic definition. Table 47 shows the directory for each platform.

Table 47. Directory for symbolic definition by platform

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/include/sqlcodes.h
HP/Sun/Linux	/opt/IBM/db2/V8.1/include/sqlcodes.h
Windows	%DB2PATH%/include/sqlcodes.h

The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.

For errors that cannot be readily mapped to any other SQL code, use SQL_RC_E1822 (SQL code = -1822).

- Code page-appropriate case conversion.
- Access to the single-byte and double-byte code pages of the current DB2 UDB database.
- Wrapper control flow information from the wrapper tracing facility.

Table 48. Public static member function behavior of the wrapper utilities class

Public static member function in C++	Public static member function in Java	Behavior
allocate	n/a	Allocate a memory block.
deallocate	n/a	Free a memory block.
report_error	WrapperException class (Java uses exceptions)	Report an error with SQL code, message, and tokens.
report_warning	reportWarning	Report a warning.
convert_to_upper	n/a	Code page-appropriate case conversion.
convert_to_lower	n/a	Code page-appropriate case conversion.
get_sb_DB_codepage	getSingleByteDBCodepage	Obtain a single-byte database code page.
get_db_DB_codepage	getDoubleByteDBCodepage	Obtain a double-byte database code page.
string_to_tokens	n/a	Tokenize a string.
trace_data	n/a	Write trace information to the trace facility.
get_db2_install_path	getDB2InstallPath	Retrieve the path to the installation.
get_db2_instance_path	getDB2InstancePath	Retrieve the path to the federated server.
get_db2_release	getDB2Release	Retrieve the version of DB2 UDB, including the Fix Pack, that the wrapper currently runs on.
fnc_entry	traceFunctionEntry	Record the entry into a function for the wrapper trace facility.

Table 48. Public static member function behavior of the wrapper utilities class (continued)

Public static member function in C++	Public static member function in Java	Behavior
fnc_exit	traceFunctionExit	Record the exit from a function for the wrapper trace facility.
fnc_data	traceFunctionData	Record the trace data by using the wrapper trace facility.
fnc_data2	traceFunctionData	Record the trace data, which includes the probe point and two data elements, by using the wrapper trace facility.
fnc_data3	traceFunctionData	Record the trace data, which includes the probe point and three data elements, by using the wrapper trace facility.
trace_error	traceException or traceError	Record the error trace data, which includes an error code and the probe points, by using the wrapper trace facility.
convert_codepage	n/a	Convert input data from the source code page to the target code page.
get_expected_conv_len	n/a	Return the number of bytes of a string that is converted from an original code page to a new code page.
get_env_lang	n/a	Return the language setting of the operating system.
change_endian2	n/a	Change the endian byte order of a double-byte character string.

Related reference:

- "WrapperUtilities class (Java)" in the *IBM DB2 Information Integrator Java API Reference for Developing Wrappers*
- "Wrapper_Utilities class (C++)" in the *IBM DB2 Information Integrator C++ API Reference for Developing Wrappers*

Chapter 10. Ensuring wrappers coexist with the environment

This section describes what you need to consider to ensure that a wrapper coexists with the rest of the environment. It includes the following topics:

- What you need to be aware of when working with system services
- How to make environment variables accessible to wrappers
- Wrapper portability

Using system services with wrappers

Because a wrapper must coexist with other federated server processes, you must be careful when using operating system services.

A complete inventory of the services that the federated server uses is not possible; it is better to assume that the federated server uses any system service and plan accordingly. This requirement applies as well to any third-party software that a wrapper could include. If it is not possible to guarantee good behavior, then some functionality must be isolated in its own process space.

The first step is to minimize the use of operating systems services wherever possible. The second step is to use those systems services routines that are provided through the wrapper utilities; it is especially important that memory management be done using these utilities.

Lastly, the wrapper must ensure that it uses services that change the system state correctly. For instance, if a wrapper uses signal handlers, it must install and remove them every time a wrapper routine is called; when the wrapper returns control to the federated server, its signal handlers cannot be left active. Another example is the use of alarms. A wrapper must restore the alarm state whenever it returns to the federated server.

Known restrictions on systems services:

- Memory management must be done through the supplied utility functions and classes.
- input/output (I/O) to stdout, stderr, stdin, cin, cout, cerr does not work.
- Windows: Wrappers must use the Win32 routine GetEnvironmentVariable(), not getenv().
- Wrappers must not use the Unix and Windows system services for tokenization, `strtok` and `strtok_r`. The wrapper interface provides replacement system services for tokenization.

Memory management (C++ only)

All memory management should be done through the wrapper utilities allocate and deallocate methods. A base class, Sqlqg_Base_Class is provided that has new and delete operators that use these methods, so classes derived from Sqlqg_Base_Class will be well-behaved.

Tokenization services (C++ only)

The Unix and Windows services `strtok` and `strtok_r` exist to break a string into parts that are based on a separator. Rather than use these, the wrapper writer must use the `string_to_token()` method of the wrapper utilities class. This method works exactly as `strtok_r` and is thread-safe.

The following code fragment illustrates the use of the `Wrapper_Utilities::string_to_token` method.

```
char*   string_to_scan = "this is a test string";
char*   scan_state = NULL;
char*   cur_token = NULL;

// Scan for the first token
cur_token = Wrapper_Utilities::string_to_token (
     string_to_scan, " ", &scan_state);
while (cur_token != NULL)
{
   // Do something useful here with the token we found

   // Get the next token; note that we pass NULL for the string
   cur_token = Wrapper_Utilities::string_to_token (
      NULL, " ", &scan_state);
}
```

Related concepts:

- "Making environment variables accessible to wrappers" on page 102

Related tasks:

- "Wrapper portablilty" on page 103

Making environment variables accessible to wrappers

The federated server controls what environment variables are available to wrappers. In order for an environment variable to be accessible by a wrapper, you must specify its value in the db2dj.ini configuration file. Table 49 shows the directory by platform where the configuration file resides on the federated server.

Table 49. Directory for configuration file on the federated server

Platform	Wrapper installation directory
AIX®	/usr/opt/db2_08_01/cfg
HP/Sun/Linux	/opt/IBM/db2/V8.1/cfg
Windows®	%DB2PATH%/cfg

The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2® Information Integrator is installed on Windows.

The federated server reads it upon startup. To change the value of an environment variable, you must stop and restart the federated server.

Entries in db2dj.ini have the following format:

```
[white space]Variable[white space]=[white space]Value[white space][eoln]
```

Note that `[white space]` is optional. Neither the variable name nor the variable value can contain white space or end of line characters. Each line in the configuration file requires the end of line character, even for the last line of the file.

The wrapper can use getenv() (on Windows platforms, GetEnvironmentVariable()) to access environment variables.

C++ coding considerations

There are restrictions on how a wrapper can use some C++ facilities, such as exceptions and Run Time Type Identification (RTTI).

A wrapper can use C++ exceptions, but the wrapper must ensure that an exception will not propagate to the federated server. Failure to catch all exceptions and turn them into return codes at the wrapper interface will result in the federated server agent process (or possibly the entire federated server instance) terminating abnormally.

The wrapper interface does not support Run Time Type Identification (RTTI). This means that some other C++ features such as dynamic cast will not be available as many compilers require RTTI for these features.

Related tasks:

- "Using system services with wrappers" on page 101
- "Wrapper portablilty" on page 103

Wrapper portablilty

For Windows, a wrapper must use the Win32 routines GetEnvironmentVariable() or GetEnvironmentStrings() routines to access environment variables rather than the Unix getenv() routine.

Related concepts:

- "Making environment variables accessible to wrappers" on page 102

Related tasks:

- "Using system services with wrappers" on page 101

Chapter 11. Documenting wrappers

Because the wrapper developer determines, in large part, what to specify on the SQL statements submitted for registration, it is advisable for the developer to let users know how to code these statements. Because the capabilities that the developer gives to a wrapper determine, to a large extent, what users can expect from data sources, it is advisable for the developer to let users know how to work with the data sources. One way to provide information for users is to compile it in a booklet or online file. This topic suggests what items to include in such a compilation and offers tips on how to treat these items.

When you document your wrapper, consider addressing the following items:

Information available at the data source
So that users can understand what sort of information they can request in their queries, you might describe:

- The nature of the information that users can retrieve from collections of data (for example, from tables, data sets, or spread sheets) at a data source
- The types of data collections available at the data source
- The nature of the retrievable information that functions at the data source can derive from stored information

Registration
In discussing registration, consider providing the following information:

- Which constructs need to be registered
- The syntax of the SQL statements with which registration will be initiated
- Which options can be coded in these statements
- What the options' allowable values are
- What the statements' parameters mean
- What the parameters' allowable values are
- How collections of data at the data source map to the parameters of the CREATE NICKNAME statement
- Which capabilities of the data source are represented by function templates

Advisories
You might include advisories—short pieces of helpful information. There are various kinds: requirements, restrictions, recommendations, hints, tips, and reminders. The examples in the following list are from IBM®'s documentation of the wrapper for table-structured file servers.

Examples of requirements

- "The column delimiter must be consistent throughout the file"
- "Statistics for nicknames of table-structured files must be updated manually by updating the SYSSTAT views."

Examples of restrictions

- "Passthru sessions are not allowed with the wrapper"
- "Files are limited to one record per line".

Examples of tips

- "The system can search sorted data files much more efficiently than non-sorted files."
- "For sorted files, you can improve performance by specifying a value or range for the key column."

Errors and warnings
: Consider also documenting the messages through which the wrapper reports errors or warns of possible problems. So that the user can look up explanations of these messages, include their associated SQLCODEs, return codes (if any), and SQLSTATE values (if any).

Users also find examples and step-by-step instructions extremely helpful.

For examples of wrapper documentation, see the *DB2® Information Integrator Data Source Configuration Guide.*

Related tasks:

- "Compiling wrappers (C++)" on page 109
- Chapter 13, "Linking wrappers (C++ only)," on page 113
- Chapter 14, "Installing wrappers," on page 117
- "Testing wrappers with valid and invalid options" on page 127
- "Compiling wrappers (Java)" on page 110

Part 4. Building, testing, and tracing wrappers

This part of the book takes you through the following tasks required to build, test, and trace a wrapper:

- Building and packaging your wrapper for deployment.
- Testing your wrapper to ensure it works as designed.

Chapter 12. Compiling wrappers

Compiling wrappers (C++)

When compiling your wrapper code, the compiler must be able to access the wrapper interface header files that are installed with the wrapper development kit. These files, along with other necessary header files, reside in the include subdirectory of the federated server.

Compiling on AIX

Use the xlC_r7 program to compile wrapper code on AIX. The following example shows the commands to use when compiling one file from the sample wrapper:

```
/usr/ibmcxx/bin/xlC_r7 -qlanglvl=ansi -qflag=i:i -qmaxmem=-1
-M -qnoansialias -qnotempinc -DSQLUNIX -g -qnamemangling=v5 -qlonglong
-I/home/inst/sqllib/include
-c sample_wrapper.C -o sample_wrapper.o
```

Use the `-qnamemangling=v5` and `-qlonglong` options if you are using the VisualAge 6.0 (or later) compiler.

For more details, see the makefile that is provided in the wrapper development kit. Table 50 shows the subdirectory by platform where the wrapper development kit is stored.

Table 50. Directory by platform for the wrapper development kit

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/samples/wrapper_sdk
HP/Sun/Linux	/opt/IBM/db2/V8.1/samples/wrapper_sdk
Windows	%DB2PATH%\samples\wrapper_sdk

The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.

Compiling on Windows

Use the CL program to compile wrapper code on Windows. The following example shows the commands to use when compiling one file from the sample wrapper:

```
C:\VC98\bin\CL -c /nologo -Zi -GZ -W3 -DNULL=0
-DWIN32 -D_X86_=1 -D_CRTAPI1=__cdecl
-D_CRTAPI2=__cdecl -D_MT -D_DLL -J -Od
-IC:\VC98\include
-IC:\sqllib\include
sample_wrapper.C /Tp -Fosample_wrapper.obj
```

Related tasks:
- Chapter 14, "Installing wrappers," on page 117
- "Testing wrappers with valid and invalid options" on page 127
- "Compiling wrappers (Java)" on page 110

Compiling wrappers (Java)

The steps that you perform to compile a wrapper depend on the language in which the wrapper is developed. This topic explains how to compile wrappers developed in Java.

Prerequisites:

You must have the Java Development Kit (JDK) version 1.3 or later. In particular, you must have the Java compiler, which is included in the JDK. The Java compiler is javac.exe on Windows and javac on UNIX.

You must have db2qgjava.jar in your class path. db2qgjava.jar is the Java archive that contains the wrapper API classes. You can add it to your system CLASSPATH or use the `-classpath` option during compilation. Table 51 shows the directory by platform where the db2qgjava.jar file is located.

Table 51. Directory by platform for Java configuration file

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/java/db2qgjava.jar
HP/Sun/Linux	/opt/IBM/db2/V8.1/java/db2qgjava.jar
Windows	%DB2PATH%\java\db2qgjava.jar

The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.

Procedure:

To compile a Java wrapper:
1. Ensure that db2qgjava.jar is in your class path.
2. Compile the custom wrapper classes as you would any other Java classes.
   ```
   javac @wrapperfiles
   ```
   The javac compiler allows you to specify various options, including:
   - The Java source files to compile. Because the Java source files have the name of the class definitions they contain, the source files for a wrapper typically include files such as UnfencedXXWrapper.java, UnfencedXXServer.java, FencedXXWrapper.java, FencedXXServer.java, and FencedXXUser.java, where XX is the name of the wrapper.
   - The location of the Java source files to compile. If the Java source files are not in the working directory or in the class path, you can specify their location in the javac command using the -sourcepath option.
   - Additions to the CLASSPATH. If you did not include db2qgjava.jar in your CLASSPATH environment variable, include it in the javac command using the -classpath option.
   - The destination directory for the compiled .class files.

   For example, you could execute the following command from a DB2 CLP window in your working directory to add the db2qgjava.jar file to your class path, compile the sample Java wrapper, and install the compiled classes in %DB2PATH%\function:

   Windows

```
javac -classpath %CLASSPATH%;%DB2PATH%\java\db2qgjava.jar
-d %DB2PATH%\function UnfencedFileWrapper.java
UnfencedFileServer.java UnfencedFileNickname.java
FencedFileWrapper.java FencedFileServer.java FencedFileNickname.java
FileConnection.java FileQuery.java FileExecDesc.java
```

UNIX

```
javac -classpath $CLASSPATH:$DB2PATH/java/db2qgjava.jar
-d %DB2PATH%/function UnfencedFileWrapper.java
UnfencedFileServer.java UnfencedFileNickname.java
FencedFileWrapper.java FencedFileServer.java FencedFileNickname.java
FileConnection.java FileQuery.java FileExecDesc.java
```

Related tasks:

- "Compiling wrappers (C++)" on page 109
- Chapter 14, "Installing wrappers," on page 117
- "Testing wrappers with valid and invalid options" on page 127

Chapter 13. Linking wrappers (C++ only)

A wrapper consists of three separate code libraries; these are the top-level library and the fenced and unfenced libraries. The shared library for the top-level library is provided as part of the wrapper development kit. The remaining two libraries must be built from your wrapper code.

The three libraries for a wrapper must conform to platform conventions for shared libraries of executable code, and they must conform to a naming convention that associates the three libraries. The following examples illustrate this.

On AIX, the sample wrapper top-level library is named ’libsample.a’. The ’lib’ prefix and ’.a’ suffix are platform conventions for shared libraries. The unfenced wrapper library is named ’libsampleU.a’. Note that this is identical to the top-level library, with the addition of the capital ’U’ just before the suffix. The fenced wrapper library is named ’libsampleF.a’.

On Windows, the sample wrapper libraries are ’db2sample.dll’, ’db2sampleU.dll’, and ’db2sampleF.dll’.

The following table lists shared library suffixes for each platform:

Table 52. Platform and supported shared library suffix

Platform	Library Suffix
AIX	.a
HP-UX	.sl
Linux	.so
Sun	.so
Windows	.dll

The unfenced wrapper library must contain the unfenced wrapper hook routine, UnfencedWrapper_Hook() and all of the code for subclasses of the following:
- Unfenced_Generic_Wrapper
- Unfenced_Generic_Server
- Unfenced_Generic_User
- Unfenced_Generic_Nickname

The fenced wrapper library must contain the fenced wrapper hook routine, FencedWrapper_Hook() and all of the code for subclasses of the following:
- Fenced_Generic_Wrapper
- Fenced_Generic_Server
- Fenced_Generic_User
- Fenced_Generic_Nickname
- Remote_Connection
- Remote_Query
- Remote_Passthru

If certain classes are not used (for example, if your wrapper uses the default implementation of the Fenced_Generic_User class), then there will be no code for those classes to be linked into the library.

Linking on AIX

The top-level library is provided in the /usr/opt/db2_08_01/lib directory of the DB2 Information Integrator installation, and is named `libdb2sqqgtop.a`. This library is already linked and only needs to be copied and renamed for the wrapper.

The fenced and unfenced libraries must be linked with a file of exported symbols; this file allows the linking program to resolve symbols that will be available when the library is loaded into the federated server environment at execution time. The exported symbol file is in the /usr/opt/db2_08_01/lib directory of the federated server and is called udbwrapper.exp.

To link the fenced and unfenced libraries on AIX, use the makeC++SharedLib_r tool. Following is an example that uses this tool to link an unfenced wrapper library:

```
/usr/lpp/xlC/bin/makeC++SharedLib_r -p 2048
-I /home/inst/sqllib/lib/udbwrapper.exp
-n UnfencedWrapper_Hook
-lpthreads -lc -lc_r
sample_wrapper.o sample_server.o sample_nickname.o
-o libsampleU.a
```

The wrapper development kit provides a makefile with more detail. The makefile is located in the /usr/opt/db2_08_01/samples/wrapper_sdk subdirectory.

Linking on Windows

The top-level library is provided in the %DB2PATH%\bin directory of the wrapper development kit, and is named `db2sqqgtop.dll`. This library is already linked and only needs to be copied and renamed for the wrapper.

The fenced and unfenced libraries must be linked with a file of exported symbols; this file allows the linking program to resolve symbols that will be available when the library is loaded into the federated server environment at execution time. The exported symbol file my be found in the %DB2PATH%\bin directory of the federated server. There are two files db2qgstp.lib and db2qg.lib. The trusted side library should be linked to db2qg.lib and the fenced side to db2qgstp.lib.

To link the fenced and unfenced libraries on Windows, use the 'link' tool. Following is an example that uses this tool to link an unfenced wrapper library:

```
C:\VC98\bin\link /OUT:db2sampleU.dll
C:\sqllib\lib\db2qg.lib
/DLL /NODEFAULTLIB /INCREMENTAL:NO
/MACHINE:i386 /SUBSYSTEM:CONSOLE
sample_wrapper.obj sample_server.obj sample_nickname.obj
```

The wrapper development kit provides a makefile with more detail. The makefile is located in the %DB2PATH%\samples\wrapper_sdk directory.

Related tasks:

- "Compiling wrappers (C++)" on page 109
- Chapter 14, "Installing wrappers," on page 117

- “Compiling wrappers (Java)” on page 110

Chapter 14. Installing wrappers

Before you can use a custom wrapper, you must install it on the federated server. The details of installing a wrapper depend on the language in which the wrapper was developed.

Installing C++ wrappers

To install a C++ wrapper:

1. Stop the federated server. You must stop the DB2 Universal Database federated server instance before you install, replace, or delete C++ wrapper libraries. Managing these libraries while DB2 UDB is running could cause DB2 UDB to crash.
2. Locate the three C++ wrapper libraries that were creating by linking the wrapper: the shared top-level library (which was included in the wrapper development kit), the fenced library, and the unfenced library.
3. Install these libraries into the appropriate directory of your DB2 UDB installation. Table 53 shows which directory applies for each platform.

Table 53. Directory for C++ wrapper installation by platform

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/lib
HP/Sun/Linux	/opt/IBM/db2/V8.1/lib
Windows	%DB2PATH%\bin

   The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.

Installing Java wrappers

You can deliver a Java wrapper in either of two formats:

- A set of .class files, which you create by compiling the Java wrapper. Typically, developers use .class files to install a wrapper only during development and testing.
- A single .jar file, which you create from the .class files using the Java Archive tool (included in the Java Development Kit). Typically, developers use a .jar file to deliver Java applications because the .jar file is easier to install. Installing the single .jar file also reduces the chance of missing files.

Installing .class files

To install the wrapper as a set of .class files:

1. Copy the all of the wrapper's .class files from your development system to the federated server. If, during the installation process, you update the CLASSPATH or if you replace existing .jar and .class files, you might need to restart DB2 UDB before using the newly installed wrapper files. For ease of installation, install the files into the following directory, depending on platform. Table 54 on page 118 shows which directory applies for each platform.

Table 54. Directory for Java Wrapper installation by platform

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/lib
HP/Sun/Linux	/opt/IBM/db2/V8.1/lib
Windows	%DB2PATH%\bin

The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.

2. Include the directory that contains the wrapper's .class files in the system CLASSPATH environment variable. The CLASSPATH environment variable usually already contains the SQLLIB\function directory.
3. Include all Java classes used by the wrapper in the system CLASSPATH environment variable.

Installing a .jar file

To install the wrapper as a .jar file:

1. Copy the wrapper .jar file from your development system to the federated server.
2. Configure the system so that the federated server can locate the wrapper .jar file. There are two ways to do this:
   - Include the path and name of the wrapper .jar file in the system CLASSPATH environment variable, or
   - Register the wrapper .jar file with the federated server using the DB2 Universal Database stored procedure `SQLJ.install_jar`. When you use `SQLJ.install_jar`, DB2 Universal Database makes a copy of the .jar file. If you update the .jar file, you must instruct DB2 Universal Database to replace its copy of the .jar file. See the DB2 Universal Database documentation for additional information.
3. Include all Java classes used by the wrapper in the system CLASSPATH environment variable.

Ensuring that Java memory sizing is sufficient

Ensure that the Java memory sizing is sufficient by modifying the JAVA_HEAP_SZ DBM environment variable for DB2 Universal Database. The minimum recommended size is 1024 for a simple wrapper, such as the sample wrapper provided in the wrapper development kit. The measure unit for JAVA_HEAP_SZ variable is 4 KB. The optimal value depends on the wrapper: how many classes are loaded, how many objects are created, and the number of concurrent connections using Java wrappers. A wrapper that loads 500–1000 classes might require a setting as high as 2048.

See the DB2 Universal Database documentation for additional information about setting the JAVA_HEAP_SZ variable.

Related tasks:

- "Installing the Develop XML Configuration File wizard" on page 120
- "Installing XML configuration files" on page 122
- "Adding data sources to the DB2 Control Center" on page 119

Chapter 15. Adding data sources to the Control Center

Adding data sources to the DB2 Control Center

You can make a custom wrapper available to users by adding it as a data source in the DB2 Control Center. Users then can choose your data source (wrapper) and specify options.

XML configuration files are used to describe data sources and their options to the DB2 Control Center. When the DB2 Control Center starts, it loads all of the data sources for which XML configuration files exist. When a user selects a data source, the Control Center displays the options that are defined in the XML configuration file for that data source.

The wrapper development kit includes a tool to assist you in creating XML configuration files.

Prerequisites:

You must have the wrapper development kit.

Procedure:

To add a custom data source to the DB2 Control Center:

1. Use the Develop XML Configuration File wizard to create an XML configuration file and associated properties files for your wrapper.
2. Install the XML configuration file. Table 55 shows the directory by platform where to install the file on the federated server.

Table 55. Directory by platform for installing the XML configuration file

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/cfg
HP/Sun/Linux	/opt/IBM/db2/V8.1/cfg
Windows	%DB2PATH%/cfg

   Table 56 shows where to find the associated properties file.

Table 56. Directory by platform for installing the XML configuration file

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/tools/en_US/wrapper_cfg
HP/Sun/Linux	/opt/IBM/db2/V8.1/tools/en_US/wrapper_cfg
Windows	%DB2PATH%\tools\en_US\wrapper_cfg

   The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.
3. If you want to support the DB2 Control Center's discovery feature so that users can discover nickname, view and server discovery information for the custom data source, then install the necessary discovery support on the federated

server. If you choose to use the built-in discovery tool, you need to install only a custom discovery stored procedure. (A graphical discovery tool is already installed with the DB2 Control Center.)

4. Restart the DB2 Control Center.

Related concepts:

- "How users add data sources to federated systems" on page 7

Related tasks:

- "Installing the Develop XML Configuration File wizard" on page 120
- "Creating XML configuration files" on page 120
- "Installing XML configuration files" on page 122

Installing the Develop XML Configuration File wizard

The Develop XML Configuration File wizard is used to create the configuration files necessary to add a wrapper (data source) to the DB2 Control Center.

The wizard is installed with the DB2 Information Integrator wrapper development kit. Table 57 shows where to look to determine if the wizard is installed on your system. Installation file for the wizard is db2qgjava.jar.

Table 57. Directory by platform for XML Configuration File wizard

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/lib/db2wrapperconfig
HP/Sun/Linux	/opt/IBM/db2/V8.1/lib/db2wrapperconfig
Windows	%DB2PATH%\bin\db2wrapperconfig.bat

The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.

Related tasks:

- "Adding data sources to the DB2 Control Center" on page 119
- "Installing the wrapper development kit" in the *IBM DB2 Information Integrator Installation Guide for Linux, UNIX, and Windows*

Creating XML configuration files

The Control Center reads XML configuration files to determine the data sources (wrappers) and options to offer users. To add a custom wrapper to the DB2 Control Center, you must create XML configuration files for the wrapper. To create XML configuration files, you use the Develop XML Configuration file wizard.

Prerequisites:

You must have the wrapper development kit installed.

You must have a Java Runtime Environment (JRE) installed. (The wizard is a Java application.)

Procedure:

To create an XML configuration file:

1. Start the Develop XML Configuration File wizard. Table 58 shows which directory from which to start the wizard.

Table 58. Directory by platform from which to start the wizard

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/lib/db2wrapperconfig
HP/Sun/Linux	/opt/IBM/db2/V8.1/lib/db2wrapperconfig
Windows	%DB2PATH%\bin\db2wrapperconfig.bat

   The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.
2. Select whether you want to create a new configuration file or modify an existing one.
3. Follow the steps provided by the wizard. The wizard prompts you for:
   - File information, including the name to use in the DB2 Control Center to identify the wrapper, the DTD path, and the XML file path.
   - Wrapper information, including the name and description to use in the DB2 Control Center to identify the wrapper's data source. Wrapper information also includes the supported operating systems and the wrapper library or class name to use for each operating system.
   - Wrapper options that users must provide to the wrapper. For each option, you can specify allowed values, the default value, and a description. You also can specify whether the option is required, whether users can edit it during creation of the wrapper, and whether users can alter it after creating the wrapper. You can also specify SQL requirements, such as whether the CREATE SERVER statement requires a user ID.
   - Server options that users must provide to the wrapper.
   - User mapping options that users must provide to the wrapper.
   - Nickname options that users must provide to the wrapper. You also can specify whether or not users can specify column definitions in the CREATE NICKNAME statement.
   - Column options that users must provide to the wrapper.
   - DB2 data types that the wrapper supports. You can specify different data types for each server.
   - Whether the DB2 Control Center supports the discover function for this wrapper. If you choose to support the discover function, there are additional steps that you must perform.
   - Environment variables that users must provide and the location where they will be set.
   - Functions templates, user-defined functions, and function mappings required by this wrapper.

   Help and infopops are available on all pages.

Results:

The wizard creates two files:

- An XML configuration file, which contains that information that you specified in the wizard. The XML configuration file is located in the directory that you specified on the second page of the wizard. The file name is based on the wrapper name. For example, if the wrapper name is `GeoDataSource`, then the XML file name is GeoDataSource.xml.

- A properties file, which contains the literal text strings that will be displayed in the DB2 Control Center for the wrapper and its options. Externalizing text strings to a properties file makes it easy to support multiple languages. You can use one configuration file with multiple properties files. The XML configuration file contains symbolic names in place of text strings; the Control Center looks up those symbolic names in the properties file that corresponds to the wrapper name and the user's language. The properties file is created in the same directory as the XML configuration file and has a name based on the wrapper name. For example, if the wrapper name is `GeoDataSource`, then the properties file name is GeoDataSource.properties.

Table 59 shows where the samples of both files are provided in the cc_plugin directory.

Table 59. Directory by platform for samples

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/samples/wrapper_sdk/cc_plugin
HP/Sun/Linux	/opt/IBM/db2/V8.1/samples/wrapper_sdk/cc_plugin
Windows	%DB2PATH%\samples\wrapper_sdk\cc_plugin

Related concepts:

- "Deciding on wrapper options" on page 33
- "Deciding on server options" on page 34
- "Deciding on nickname and column options" on page 29
- "Deciding on user mapping options" on page 35
- "How users add data sources to federated systems" on page 7

Related tasks:

- "Determining the head expressions that the data source can accept" on page 37
- "Determining the predicates that the data source can accept" on page 37
- "Determining the joins that the data source can accept" on page 38
- "Determining the functions that the data source can accept" on page 38
- "Adding data sources to the DB2 Control Center" on page 119

Installing XML configuration files

The DB2 Control Center determines which wrappers (data sources) and options to present to users by reading XML configuration files. After you create the XML configuration files for the wrapper, you must install those files on the federated server from which the DB2 Control Center can load them. To add a data source to the DB2 Control Center, you must install the XML configuration files for the data source on the federated server.

Prerequisites:

You must have access to the set of XML configuration and properties files for the wrapper.

You must have access to the federated server.

Procedure:

To install the XML configuration file for a wrapper:

1. Transfer the wrapper XML configuration file and associated properties files to the federated server.
2. Move the XML configuration file (wrappername.xml). Table 60 shows what directory to move the file to.

Table 60. Directory by platform that you move the XML configuration file to

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/cfg
HP/Sun/Linux	/opt/IBM/db2/V8.1/cfg
Windows	%DB2PATH%\cfg

   The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.

3. Move each properties file (wrappername.properties) to the (wrapper_cfg) directory. For example, if you provide US English support for your wrapper-specific additions to the DB2 Control Center, then move the US English properties file for the wrapper. Table 61 shows what directory to move the file to.

Table 61. Directory by platform that you move properties file to

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/tools/en_US/wrapper_cfg
HP/Sun/Linux	/opt/IBM/db2/V8.1/tools/en_US/wrapper_cfg
Windows	%DB2PATH%\Tools\en_US\wrapper_cfg

4. Restart the DB2 Control Center.

When the DB2 Control Center is started, it loads options for all of the data sources for which an XML configuration file and properties file are installed. Your data source is presented as an option, along with the standard IBM data sources.

Related concepts:

- "How users add data sources to federated systems" on page 7

Related tasks:

- Chapter 14, "Installing wrappers," on page 117
- "Creating XML configuration files" on page 120

Supporting discovery in the DB2 Control Center

The DB2 Control Center can discover features of data sources, such as servers, view, and nicknames. When you develop a custom wrapper, you can choose to provide support for the DB2 Control Center's discovery feature. When you provide discovery support, users can discover nickname, view and server information for the custom data source in the same way that they can for standard data sources.

Prerequisites:

On the development system, you must have the wrapper developer's kit.

On the federated server, you must have a Java Runtime Environment (JRE).

Procedure:

To support discovery for a custom wrapper (data source):

1. Create an XML configuration file that specifies the discovery function and options to use for your wrapper:
   a. Start the Develop XML Configuration File wizard.
   b. If an XML configuration file exists for the wrapper, choose to modify that file. If an XML configuration file does not exist, choose to create a new one. Follow the instructions in
   c. On the **Specify the discover function requirements** page of the wizard, select one of the two options to support the discover function:
      - **Support the discover function using a built-in concrete Java class** uses the simple Java discovery tool that is installed with the DB2 Control Center and the wrapper development kit. (In the DB2 Control Center installation, the tool is provided as the Java file db2WrapperDiscovery.jar. In the wrapper development kit, it is called db2WrapperDiscoverySDK.jar. The contents of the two files are identical.)
      - **Support the discover function using a custom Java class** uses a custom Java class that you provide.
   d. On the same page, add any custom or user interface options for the discovery class.
   e. Finish the wizard.
2. If you selected the built-in concrete Java class, you must create a stored procedure that returns the information that you want discovered from your wrapper's data source. Table 62 shows the directory for the wrapper development kit that includes a Java stored procedure that you can use as an example.

Table 62. Directory by platform for Java stored procedure

Platform	Wrapper installation directory
AIX	/usr/opt/db2_08_01/samples/wrapper_sdk/cc_plugin/sample.java
HP/Sun/Linux	/opt/IBM/db2/V8.1/samples/wrapper_sdk/cc_plugin/sample.java
Windows	%DB2PATH%\samples\wrapper_sdk\cc_plugin\sample.java

   The default Windows directory path is C:\Program Files\IBM\SQLLIB. %DB2PATH% is the environment variable that is used to specify the directory path where DB2 Information Integrator is installed on Windows.

   The sample stored procedure is an example in Java of how the build-in discovery works. You can use a C stored procedure, a SQL stored procedure, or any other stored procedure to implement your discovery function. To create a custom stored procedure from the sample provided in cc_plugin:

   a. Using sample.java as a guide, create a stored procedure that will return the information that you want discovered for your data source.
   b. Update the sample makefile so that it refers to your Java stored procedure.
   c. Use the makefile to compile the Java stored procedure.
   d. Update the sample.db2 script so that it will load the stored procedure into the DB2 Universal Database instance that is your federated server.

   e. Run your updated sample.db2 script to install the stored procedure in the federated server.
3. If you selected a custom Java class, then you must provide the class and any supporting stored procedures that it requires. Install the custom Java class and any other necessary components on the federated server.
4. Install the XML configuration files on the federated server.
5. Restart the DB2 Command Center.

Results:

After you install the wrapper, the wrapper's XML configuration files, and the stored procedure, DB2 Control Center users will see the custom data source as an option and will be able to perform discovery for that data source. The DB2 Control Center reads the XML configuration file to determine if it should support discovery for this data source and, if so, which Java class to use for the discovery tool.

If you choose the built-in concrete Java class for discovery, then you do not need to install any other discovery tools. The DB2 Control Center will use its built-in Java tool in combination with your custom stored procedure to perform discovery.

Related tasks:
- "Installing the Develop XML Configuration File wizard" on page 120
- "Creating XML configuration files" on page 120
- "Installing XML configuration files" on page 122
- "Adding data sources to the DB2 Control Center" on page 119

Chapter 16. Testing wrappers

The following sections describe how to test wrappers.

Using registration DLL statements to test wrappers

SQL statements such as CREATE SERVER and ALTER SERVER can be used for testing a wrapper. For example, to test your implementation of the wrapper base class, you can run CREATE WRAPPER. To test your implementation of the Server base classes, you can run CREATE SERVER.

It is up to you to determine what information to specify in SQL statements for purposes of testing and registration. To make such a determination, you need to understand these statements' parameters. To acquire this understanding, see the *SQL Reference* for DB2® Universal Database Version 8.

Related tasks:

- "Testing wrappers with valid and invalid options" on page 127
- "Creating trace information from wrappers" on page 130

Testing wrappers with valid and invalid options

It is advisable to test options information for the following conditions:

- A specified option should be valid for the SQL statement that contains the option. Otherwise, the wrapper should return SQLCODE SQL1881N.
- The value to which an option has been set should be valid for this option. Otherwise, the wrapper should return SQLCODE SQL1882N.

  In some cases, an option value is invalid because it is inconsistent with another option value. For example, suppose that a data source encodes query results when the user sets an option called ENCRYPT to Y (yes, encrypt this result set) and an option called ENCRYPTION_KEY to a string of symbols required by the encryption program. Clearly, when ENCRYPT is set to N (no, do not encrypt this result set), the string of symbols (or any value at all, for that matter) would be invalid for ENCRYPTION_KEY.
- Options required on an SQL statement should be specified in the statement. If a required option is missing, the wrapper should return SQLCODE SQL1883N.
- An option should be specified only once on the same SQL statement. If it is specified more than once, DB2 returns SQLCODE SQL1884N.
- If the user uses an ALTER [CONSTRUCT] statement (for example, ALTER SERVER) to add an option that has already been defined, DB2 returns SQLCODE SQL1885N.
- A user can use an ALTER [CONSTRUCT] statement (for example, ALTER SERVER) to update or delete a value of an option only if the option has already been set to a value. If the option has not been set to a value, DB2 returns SQLCODE SQL1886N.

Related concepts:

- "Using registration DLL statements to test wrappers" on page 127

Related tasks:

- "Creating trace information from wrappers" on page 130

Chapter 17. Tracing wrappers

The wrapper tracing facility logs control flow and troubleshooting information from your wrapper. The following sections describe the wrapper tracing facility and how to create trace information from your wrapper

Wrapper trace facility

The wrapper trace facility records control flow information, such as function entry points and function exit points, from the wrappers that you develop. You can use this information to identify potential problems when you are developing your wrapper and to solve problems that you might encounter after your wrapper is deployed.

To obtain this troubleshooting information, you call the tracing member functions from the areas that you specify in your wrapper. You can then use the **db2trc** command to start and to control the trace operation.

The wrapper utilities class provides a set of member functions that you can use in the wrapper code to retrieve the function entry and exit points and the data from the error trace. The following table describes the purpose of these member functions in.

Table 63. Tracing member functions for the wrapper utilities class

Member function in C++	Member function in Java™	Purpose
fnc_entry	traceFunctionEntry	Record the entry into a function.
fnc_exit	traceFunctionExit	Record the exit from a function.
fnc_data	traceFunctionData	Record the data trace, which includes the probe point and a single data element.
fnc_data2	traceFunctionData	Record the data trace, which includes the probe point and two data elements, if needed.
fnc_data3	traceFunctionData	Record the data trace, which includes the probe point and three data elements, if needed.
trace_error	traceError or traceException	Record the error trace, which includes an error code and the probe points.

After you enter the tracing member functions in your wrapper code, you can use the **db2trc** command to start the trace facility. The trace facility collects the wrapper control flow information. To collect this data from your wrapper, you must issue the **db2trc** command with a trace record component ID of 135. A component ID of 135 is specific to custom wrappers.

Note: The component ID for external wrappers might change. See the *DB2 Information Integrator Release Notes* for the latest information.

Related tasks:

- "Creating trace information from wrappers" on page 130

Related reference:

- "Example of wrapper trace facility" on page 131

Creating trace information from wrappers

You can use the trace facility to log control flow information and to obtain troubleshooting information from your wrapper. The wrapper utilities class provides member functions that collect control flow records from your wrapper. Use the **db2trc** command to extract these control flow records and to format this information into readable text.

Procedure:

To create trace information from your wrapper:

1. Issue the **db2trc** command with an adequate buffer size.

   For example:

   ```
   db2trc on -l 8M
   ```
2. After you run the trace facility for an appropriate length of time, write the current contents of the trace buffer to a file.

   For example:

   ```
   db2trc dump trc.dmp
   ```

   This command writes the trace information into an output file named trc.dmp in the current directory.
3. Turn off the trace facility by issuing the following command:

   ```
   db2trc off
   ```
4. Write the contents of the trace output to a file that contains error code and trace flow information from your wrapper.

   For example:

   ```
   db2trc flw  -m *.*.135.*.* trc.dmp trc.flw
   ```

   You must specify a mask option of `*.*.135.*.*` to retrieve only the trace records that correspond to external wrappers. The value of 135 is the component ID for external wrappers.

   Note: The component ID for external wrappers might change. See the *DB2 Information Integrator Release Notes* for the latest information.
5. Write the contents of the trace output to a file that contains the trace data from your wrapper in chronological order.

   For example:

   ```
   db2trc flw  -m *.*.135.*.* trc.dmp trc.fmt
   ```

Related concepts:

- "Wrapper trace facility" on page 129

Related reference:

- “db2trc - Trace Command” in the *Command Reference*
- “Example of wrapper trace facility” on page 131

Example of wrapper trace facility

This topic provides examples that show you how to call tracing member functions by using the wrapper utilities class in your wrapper code. This topic also provides example trc.flw and trc.fmt files that contain control flow information from the tracing member functions in wrapper code. You generate trace files by issuing the **db2trc** command.

This example uses the C++ method names. If you are developing your wrapper in Java, substitute the corresponding Java method name.

The following example shows an UnfencedWrapper_Hook function that uses the wrapper utilities class to call tracing member functions. This example obtains trace information for two data elements that are indicated by the fnc_data2 member function.

```
extern "C" UnfencedWrapper* UnfencedWrapper_Hook()
{

    #define FUNC_ID 1
    UnfencedWrapper* wrapper=NULL;
    sqlint32 rc=0;
    const char* fName = "UnfencedWrapper_Hook";
    Wrapper_Utilities::fnc_entry(FUNC_ID, fName);
    Wrapper_Utilities::fnc_data2(FUNC_ID, fName, 10,
         strlen("First Function"), "First Function", sizeof(rc), &rc);

    wrapper = new(&rc) Sample_Wrapper(&rc);

    if( (rc) || (wrapper == NULL) )
    {
      Wrapper_Utilities::trace_error(FUNC_ID, fName,
                           30, sizeof(rc), &rc);

      if (wrapper != NULL )
      {
          delete wrapper;
          wrapper = NULL;
      }

    }

    Wrapper_Utilities::fnc_exit(FUNC_ID, fName, rc);
    return wrapper;
}
```

Figure 10. Wrapper code with tracing member functions

You can then issue the **db2trc** command to start the trace facility.

The following examples show the contents of the trc.flw and trc.fmt files after you run the trace facility.

An example of a trc.flw file:

The trc.flw file contains the wrapper tracing error codes, function entry points, function exit points, and data trace points.

`probe 0` indicates a reference to the function name that calls the tracing facility. Function names are not specified in the trc.flw file.

```
pid = 2316 tid = 2292 node = 0

1          Func~1 entry
2          Func~1 data [probe 0]
3          Func~1 data [probe 0]
4          Func~1 data [probe 10]
5          Func~1 data [probe 0]
6          Func~1 exit
```

Figure 11. trc.flw file

An example of a trc.fmt file:

The trc.fmt file contains the actual function names that are indicated by `probe 0` in the trc.flw file.

Lines 1 and 6 in the following trc.fmt file indicate the function entry and exit points, respectively. Lines 2, 3, and 5 indicate a reference to the UnfencedWrapper_Hook function that calls the tracing facility from the wrapper code. Line 4 shows the data trace and data elements.

```
1 entry DB2 External Wrappers Func~1 fnc (1.3.135.1.0)
  pid 2316 tid 2292 cpid -1 node 0 sec 0 nsec 0

2 data DB2 External Wrappers Func~1 fnc (3.3.135.1.0.0)
  pid 2316 tid 2292 cpid -1 node 0 sec 0 nsec 527 probe 0
  bytes 28
  Data1  (PD_TYPE_HEXDUMP,20) Hexdump:
  556E 6665 6E63 6564 5772 6170 7065 725F 486F 6F6B   UnfencedWrapper_Hook

3 data DB2 External Wrappers Func~1 fnc (3.3.135.1.0.0)
  pid 2316 tid 2292 cpid -1 node 0 sec 0 nsec 639 probe 0
  bytes 28
  Data1  (PD_TYPE_HEXDUMP,20) Hexdump:
  556E 6665 6E63 6564 5772 6170 7065 725F 486F 6F6B    UnfencedWrapper_Hook

4 data DB2 External Wrappers Func~1 fnc (3.3.135.1.0.10)
  pid 2316 tid 2292 cpid -1 node 0 sec 0 nsec 696 probe 10
  bytes 34

  Data1  (PD_TYPE_HEXDUMP,14) Hexdump:
  4669 7273 7420 4675 6E63 7469 6F6E         First Function

  Data2  (PD_TYPE_HEXDUMP,4) Hexdump:
  0000 0000                                  ....

5 data DB2 External Wrappers Func~1 fnc (3.3.135.1.0.0)
  pid 2316 tid 2292 cpid -1 node 0 sec 0 nsec 2644 probe 0
  bytes 28

  Data1  (PD_TYPE_HEXDUMP,20) Hexdump:
  556E 6665 6E63 6564 5772 6170 7065 725F 486F 6F6B   UnfencedWrapper_Hook

6 exit DB2 External Wrappers Func~1 fnc (2.3.135.1.0)
  pid 2316 tid 2292 cpid -1 node 0 sec 0 nsec 2737 rc = 0
```

Figure 12. trc.fmt file

Related concepts:

- "Wrapper trace facility" on page 129

Related tasks:

- "Creating trace information from wrappers" on page 130

Related reference:

- "db2trc - Trace Command" in the *Command Reference*

Glossary

C

collaboration. An occurrence of a sequence of operations that realizes a use case scenario. Also known as user-to-user. It typically involves collaboration between two or more components. As an example, consider the scenario of updating customer details in a client-server system. There is a sequence of operations in which the graphical user interface (GUI) component displays a window, calls a data server component with a request for data, displays the customer details (and amends them), calls the data server to perform an update. This whole pattern of component operations and exchanges between components is a collaboration that ″realizes″ the scenario.

composition. This is the process of creating (or composing) an XML document from data in DB2 tables. Elements in the generated XML document are created from fields in one or more DB2 tables. And, the XML document can be stored in DB2 or outside of DB2, in the file system and MQSeries message queues.

D

decomposition. Also known as shredding. This is the process of storing an XML document in DB2. The XML document is broken apart (or shredded) and the elements are stored as fields in one or more DB2 tables.

DB2 Administrator. A person responsible for administrative tasks such as access authorization and content management. Administrators can also grant levels of authority to users.

decomposition. Also known as shredding. This is the process of storing an XML document in DB2. The XML document is broken apart (or shredded) and the elements are stored as fields in one or more DB2 tables.

E

enterprise java bean. An enterprise java bean is a Java component that can be combined with other enterprise beans and other Java components to create a distributed application. There are two types of enterprise beans: an entity bean and a session bean.

extended enterprise applications. Applications that integrate programmatic interactions among organizations. Also known as business-to-business applications.

F

fenced. Pertaining to a type, or characteristic, of a procedure, user-defined function, or federated wrapper that is defined to run in a separate process from the database manager. When this type of object is run (using the fenced clause), the database manager is protected from modifications by the object.

H

heterogeneous systems. A collection of dissimilar systems with a range of diverse computing resources that can be local to one another or geographically distributed.

hypertext transfer protocol (http). The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems.

I

information aggregation applications. Applications where tools extract information from other data sources. Also known as user-to-data.

informational data. Data that is extracted from the operational data and then transformed for decision making. See also operational data.

J

Java 2 Platform, Enterprise Edition (J2EE). A platform that offers a multitiered distributed application model, the ability to reuse components, integrated Extensible Markup Language (XML)-based data interchange, a unified security model, and flexible transaction control.

informational data. Data that is extracted from the operational data and then transformed for decision making. See also operational data.

L

legacy data. Data that is produced from decades of information gathering and data analysis, that you already have and use. Most often, this takes the forms of records in an existing database on a system in current use.It is generally information that exists in local or proprietary databases, safely tucked away in data management systems, that is usually not available to an enterprise system.

N

namespace table. A Namespace Table (NST) resource defines the mapping from DB2 XML Extender DTDIDs to XML Schema (XSD) namespaces and locations. Namespaces enable you to mix, in one XML document, element (and sometimes attribute) names from more than one XML vocabulary.

nickname. (1) In a federated system, an identifier that is used in a query to refer to an object at a data source. The objects that nicknames identify are referred to as data source objects. Examples of data source objects include tables, views, synonyms, table-structured files, and search algorithms. (2) A name that is defined in DB2 Information Integrator to represent a physical database object (such as a table or stored procedure) in a non-DB2 relational database.

S

scenario. An instance of a use case. That is, a scenario is an execution of a use case under well-specified assumptions. A scenario is realized in a particular system by a collaboration.

self-service applications. Also known as User-to-Business applications where users are interacting with enterprise transactions and data

semistructured data. Data that can include spreadsheets, address books, configuration parameters, financial transactions, or technical drawings. The Structured Query Language (SQL) works well with structured data.

structured data. Data that can include spreadsheets, address books, configuration parameters, financial transactions, or technical drawings. The Structured Query Language (SQL) works well with structured data.

U

unfenced. Pertaining to a type, or characteristic, of a procedure, user-defined function, or federated wrapper that is defined to run in the database manager process. When this type of object is run (using the not fenced clause), the database manager is not protected from changes made by this object.

unstructured data. Any data that is stored unorganized, and possibly outside of a traditional database. Some examples of this data are text, audio, video, fax, image, or graphics.

use case. An identifiable and externally observable behavior within a particular system. It is a pattern of usage that is initiated by an actor or user, and that performs or aims to perform some useful work. A use case represents a dialog between an actor and the system. For example, "Draw funds from checking account" is a use case.

W

WORF. Web object runtime framework. The runtime engine that supports the DB2 Web services provider.

wrapper. In a federated system, the mechanism that the federated server uses to communicate with and retrieve data from the data sources. To implement a wrapper, the federated server uses routines stored in a library called a wrapper module. These routines allow the federated server to perform operations such as connecting to a data source and retrieving data from it iteratively. The DB2 Universal Database federated instance owner uses the CREATE WRAPPER statement to register a wrapper for each data source that is to be included in the federated system.

X

XSD. Extensible Markup Language (XML) schema definition. A language for describing XML files that contain schema.

Accessibility

Accessibility features help users with physical disabilities, such as restricted mobility or limited vision, to use software products successfully. The following list specifies the major accessibility features in DB2® Version 8 products:

- All DB2 functionality is available using the keyboard for navigation instead of the mouse. For more information, see "Keyboard input and navigation."
- You can customize the size and color of the fonts on DB2 user interfaces. For more information, see "Accessible display."
- DB2 products support accessibility applications that use the Java™ Accessibility API. For more information, see "Compatibility with assistive technologies" on page 138.
- DB2 documentation is provided in an accessible format. For more information, see "Accessible documentation" on page 138.

Keyboard input and navigation

Keyboard input

You can operate the DB2 tools using only the keyboard. You can use keys or key combinations to perform operations that can also be done using a mouse. Standard operating system keystrokes are used for standard operating system operations.

For more information about using keys or key combinations to perform operations, see Keyboard shortcuts and accelerators: Common GUI help.

Keyboard navigation

You can navigate the DB2 tools user interface using keys or key combinations.

For more information about using keys or key combinations to navigate the DB2 Tools, see Keyboard shortcuts and accelerators: Common GUI help.

Keyboard focus

In UNIX® operating systems, the area of the active window where your keystrokes will have an effect is highlighted.

Accessible display

The DB2 tools have features that improve accessibility for users with low vision or other visual impairments. These accessibility enhancements include support for customizable font properties.

Font settings

You can select the color, size, and font for the text in menus and dialog windows, using the Tools Settings notebook.

For more information about specifying font settings, see Changing the fonts for menus and text: Common GUI help.

Non-dependence on color

You do not need to distinguish between colors in order to use any of the functions in this product.

Compatibility with assistive technologies

The DB2 tools interfaces support the Java Accessibility API, which enables you to use screen readers and other assistive technologies with DB2 products.

Accessible documentation

Documentation for DB2 is provided in XHTML 1.0 format, which is viewable in most Web browsers. XHTML allows you to view documentation according to the display preferences set in your browser. It also allows you to use screen readers and other assistive technologies.

Syntax diagrams are provided in dotted decimal format. This format is available only if you are accessing the online documentation using a screen-reader.

Related concepts:

- "Dotted decimal syntax diagrams" in the *Infrastructure Topics (DB2 Common Files)*

Related tasks:

- "Keyboard shortcuts and accelerators: Common GUI help"
- "Changing the fonts for menus and text: Common GUI help"

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

Index

Contacting IBM

To contact IBM customer service in the United States or Canada, call 1-800-IBM-SERV (1-800-426-7378).

To learn about available service options, call one of the following numbers:

- In the United States: 1-888-426-4343
- In Canada: 1-800-465-9600

To locate an IBM office in your country or region, see the IBM Directory of Worldwide Contacts on the Web at www.ibm.com/planetwide.

Product information

Information about DB2 Information Integrator is available by telephone or on the Web.

If you live in the United States, you can call one of the following numbers:

- To order products or to obtain general information: 1-800-IBM-CALL (1-800-426-2255)
- To order publications: 1-800-879-2755

On the Web, go to www.ibm.com/software/data/integration/db2ii/support.html. This site contains the latest information about:

- The technical library
- Ordering books
- Client downloads
- Newsgroups
- Fix packs
- News
- Links to Web resources

Comments on the documentation

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 Information Integrator documentation. You can use any of the following methods to provide comments:

- Send your comments using the online readers' comment form at www.ibm.com/software/data/rcf.
- Send your comments by e-mail to comments@us.ibm.com. Include the name of the product, the version number of the product, and the name and part number of the book (if applicable). If you are commenting on specific text, please include the location of the text (for example, a title, a table number, or a page number).

Printed in USA

SC18-9174-00

Spine information:

IBM DB2 Information Integrator   Wrapper Developer's Guide   Version 8.2