Teiid - Scalable Information Integration

1

Teiid Connector Developer's Guide

6.0.0 GA

Connecting to Your Enterprise Information System

The Teiid System offers your organization a way to manage and describe the information across your disparate enterprise information systems. You can even integrate these enterprise information systems into a single, complete data access solution using the Teiid Server.

1.1. The Teiid System

The entire Teiid System is comprised of several interconnected products and services:

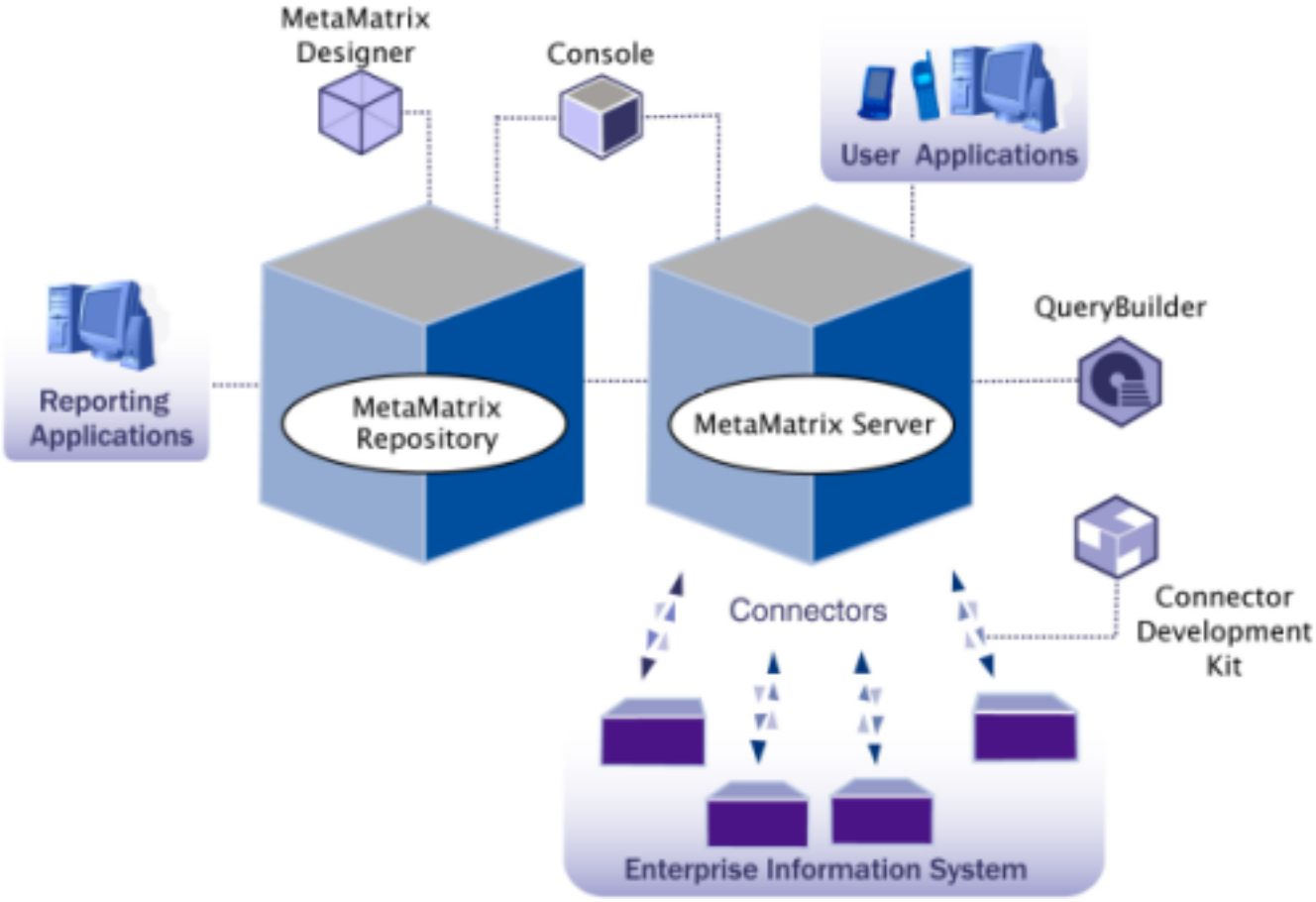

Figure 1.1. Teiid System

The Teiid System, when used in its totality, enables your end user applications to process queries that select (and even update) data from one or more of your enterprise information sources, regardless of the native physical data storage method used by each enterprise information system. This means that a single query can access, reference, and return results from multiple integrated data sources.

Within the Teiid System, the Teiid products (including the Teiid Designer, the Teiid Server), enable you to create and manage metadata models: representations describing the nature and content of your enterprise information systems.

Once captured, this valuable metadata can searched, analyzed, and applied by applications throughout your enterprise.

Connectors in the Teiid System

In the Teiid System, a connector handles all request-and-response related communications between the data tier of the Teiid Server and the individual enterprise information sources, which can include databases, data feeds, flat files, or any other entity you have modeled

In the Teiid Server, a connector is used to:

- Translate a Teiid-specific command into a native command.
- Execute the command.
- Return batches of results to the Teiid Server.

The Teiid Server is responsible for reassembling the results from one or more connectors into an answer for the user's command.

For a more detailed workflow, see the chapter *"Connector API."*

2.1. Do You Need a New Connector?

Teiid can provide several connectors for common enterprise information system types. If you can use one of these enterprise information systems, you do not need to develop a custom one. Instead, you can contact your Teiid Technical Account Manager and ask about purchasing the connector you need.

Teiid offers the following connectors:

- *JDBC:* Connects to many relational databases. The JDBC Connector is validated against the following database systems: Oracle, Microsoft SQL Server, IBM DB2, MySQL and Sybase. In addition, the JDBC Connector can often be used with other 3rd-party drivers and provides a wide range of extensibility options to specialize behavior against those drivers.
- *Text:* Connects to ASCII text files.
- *XML* Connects to XML files on disk or by invoking Web services on other enterprise systems.

If your enterprise information system can use one of these connectors, you do not need to develop your own. Instead, you can contact Teiid about acquiring the connector you need.

2.2. Required Items to Write a Custom Connector

To write a connector, follow this procedure:

1. Gather all necessary information about your Enterprise Information System (EIS). You will need to know:

- API for accessing the system
- Configuration and connection information for the system
- Metadata
- Required properties for the connector, such as URL, user name, etc.
- The CDK development kit (jars and tools).

2. Implement the required interfaces defined by the Connector API.
   - Connector – starting point.
   - Connection – represents a connection to the source.
   - ConnectorCapabilities – specifies what kinds of commands your connector can execute
   - Execution (and sub-interfaces) – specifies how to execute each type of command
3. Test your connector with Connector Development Kit (CDK) test utilities.
4. Deploy your connector type into a Teiid Server using the Teiid Console.
   - Create your connector type definition file. Import the connector type definition file
   - Create a connector binding using the connector type
   - Deploy a Virtual Database with metadata corresponding to your EIS
5. Execute queries via the Teiid JDBC API or QueryBuilder

This guide covers how to do each of these steps in detail. It also provides additional information for advanced topics, such as connection pooling, streaming large objects, and transactions. For a sample connector code, please check the wiki pages at *Teiid community* [http://teiid.org]

Connector API

3.1. Overview

A component called the Connector Manager is controlling access to your connector. This chapter reviews the basics of how the Connector Manager interacts with your connector while leaving reference details and advanced topics to be covered in later chapters.

A custom connector must implement the following interfaces to connect and query an enterprise Data Source. These interfaces are in package called *com.metamatrix.data.api:*

- *Connector* - This interface is the starting point for all interaction with your connector. It allows the Connector Manager to obtain a connection and perform lifecycle events.
- *Connection* - This interface represents a connection to your data source. It is used as a starting point for actual command executions. Connections provided to the Connector Manager will be obtained and released for each command execution. Teiid provides for extensible automatic connection pooling, as discussed in the *Connection Pooling* chapter.
- *ConnectorCapabilities* - This interface allows a connector to describe the execution capabilities of the connector. Teiid provides a base implementation of this class called BasicConnectorCapabilities. You can either extend this basic implementation or implement your own implementation.
- *Execution (and sub-interfaces)* - These interfaces represent a command execution with your Connector. There is a sub-interface for executing each kinds of command: query, update, and procedure. Your connector can specify via the ConnectorCapabilities which of these command types it can handle.
- *Batch* - This interface represents a batch of results being sent to the Teiid Server from the custom connector. Teiid provides a default implementation of this class called BasicBatch.

The most important interfaces provided by Teiid to the connector are the following:

- *ConnectorEnvironment* – an interface describing access to external resources for your connector.
- *ConnectorLogger* – an interface for writing logging information to Teiid logs.
- *SecurityContext / ExecutionContext* – interfaces defining the security information and execution context available to the connector when executing a command.

3.2. Connector Lifecycle

3.2.1. Initialization

A Connector will be initialized one time via the initialize() method, which passes in a *ConnectorEnvironment* object provided by the *Connector Manager* . The *ConnectorEnvironment provides* the following resources to the connector:

- Configuration properties – name / value pairs as provided by the connector binding in the Teiid Console
- Logging – ConnectorLogger interface allows a Connector to log messages and errors to Teiid's log files.
- Runtime metadata – access to the runtime metadata of the model deployed in the Teiid Server for your physical source
- Large object replacement – ability to stream the results of large values (such as blobs and clobs) through the Teiid system
- Type facility – an interface defining runtime datatypes and type conversion facility.

3.2.2. Starting and Stopping

Other methods on the connector allow the Connector Manager to start() or stop() the connector. Typically the connector is started or stopped in response to system startup, system shutdown, or an administrator changing these states on connector bindings in the Teiid Console or through Admin API. A connector should perform whatever actions are necessary in these methods to create or destroy all connector states, including connections to the actual physical source.

3.3. Connections to Source

3.3.1. Obtaining connections

The connector must implement the getConnection() method to allow the Connector Manager to obtain a connection. The getConnection() method is passed a SecurityContext, which contains information about the context in which this query is being executed.

The SecurityContext contains the following information:

- User name
- Virtual database name
- Virtual database version
- Trusted token

The trusted token is used to pass security information specific to your application through the Teiid Server. The client can pass the trusted token when they connect via JDBC. This token is

then passed to the Membership Service and may be created, replaced, or modified at that time. In some cases, you may wish to provide a customer Membership Service implementation to handle security needs specific to your organization. For more information on implementing a custom Membership Service, contact Teiid technical support.

3.3.2. Releasing Connections

Once the Connector Manager has obtained a connection, it will use that connection only for the lifetime of the request. When the request has completed, the release() method will be called on the connection.

In cases (such as when a connection is stateful and expensive to create), connections should be pooled. Teiid provides an extensible connection pool for this purpose, as described in chapter *Connection Pooling*.

3.4. Executing Commands

3.4.1. Execution Modes

The Connector API uses a connection to obtain an execution interface for the command it is executing. Connectors may support any subset of the available execution modes. The execution modes are defined by constants in the ConnectorCapabilities.EXECUTION_MODE class. The following execution modes are available:

Table 3.1. Types of Execution Modes

Execution Mode	Execution Interface	Command interface(s)	Description
Synchronous Query	`SynchQueryExecution`	`IQuery`	A query, corresponding to a SQL SELECT statement
Update	`UpdateExecution`	`IInsert`, `IUpdate`, `IDelete`	An insert, update, or delete, corresponding to a SQL INSERT, UPDATE, or DELETE command
Procedure Execution	`ProcedureExecution`	`IProcedure`	A procedure execution that may return a result set and/or output values.
Asynchronous Query	`AsynchQueryExecution`	`IQuery`	Polled asynchronous execution of a SQL SELECT statement
Batched Update	`BatchedUpdatesExecution`	`ICommand[]`	Execute multiple commands in a batch
Bulk Insert	`BatchedUpdatesExecution`	`IInsert`	Insert a large set of data using the same INSERT command

Following is a class diagram further defining the relationships between the execution interfaces:

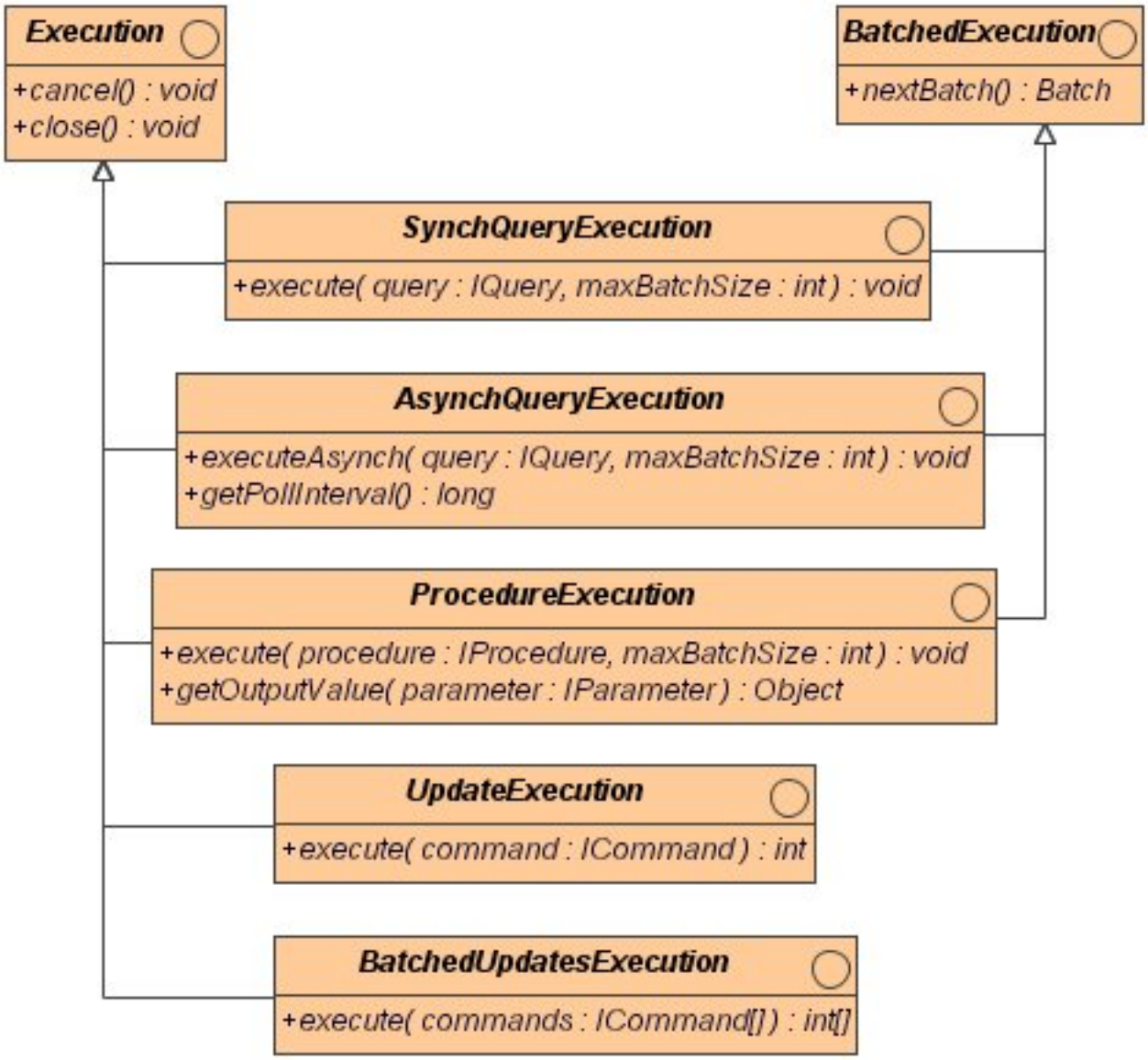

Figure 3.1. Execution Interfaces Class Diagram

All of the execution interfaces extend the base execution interface that defines how executions are cancelled and closed. SynchQueryExecution, AsynchQueryExecution, and ProcedureExecution all extend the BatchedExecution interface, which defines how batched results are returned from an execution.

3.4.2. Synchronous Query Execution

Most commands executed against connectors are queries. Queries correspond to the SELECT statement in SQL. The actual queries themselves are sent to connectors in the form of a set of objects, which are further described in Chapter *Command Language*.

The following diagram represents the typical sequence of events when executing a query:

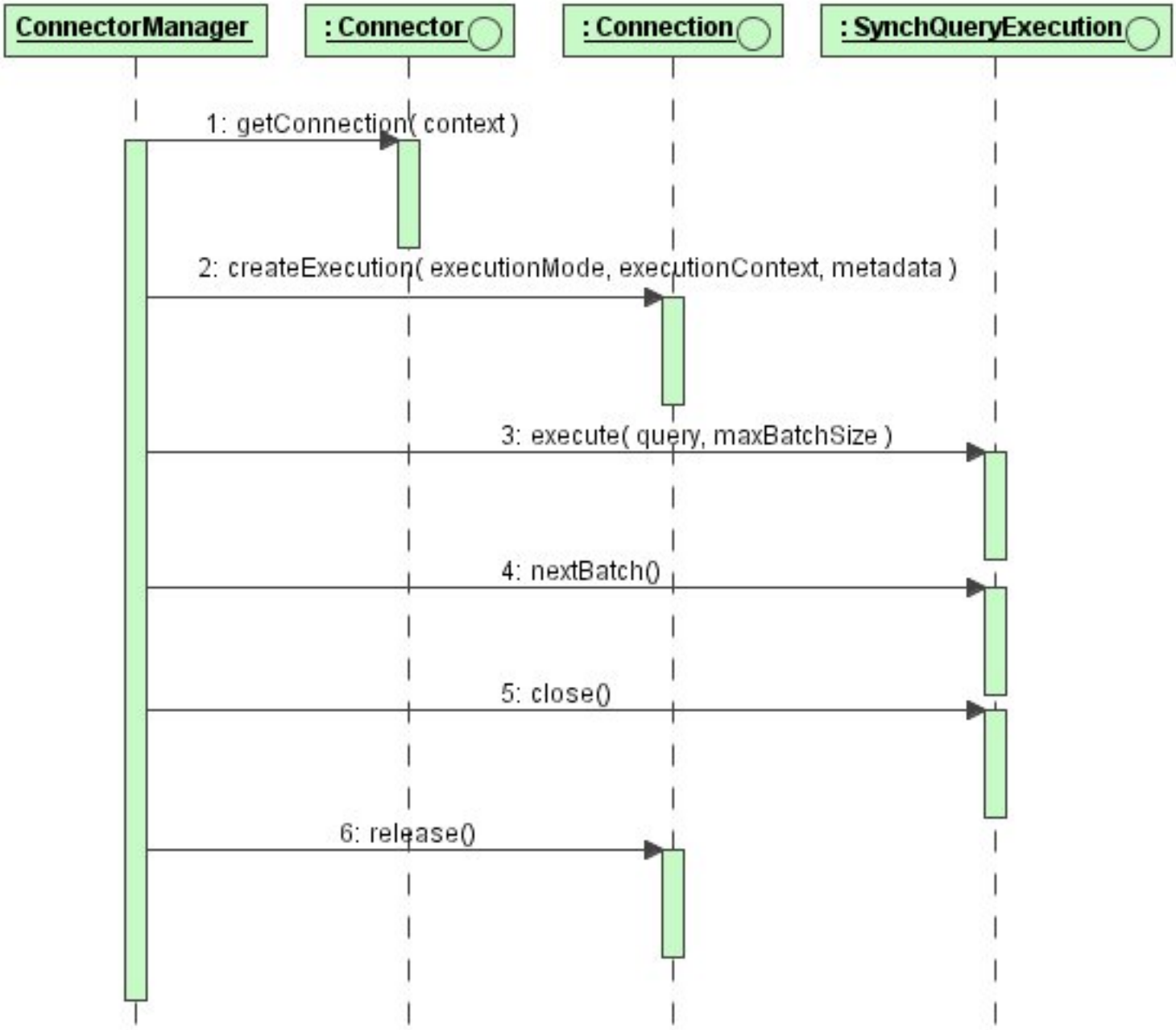

Figure 3.2. Query Execution Sequence Diagram

While the command is being executed, the connector retrieves results in batches via the BatchedExecution interface. Each time nextBatch() is called, the SynchQueryExecution implementation should return a batch of no more than maxBatchSize records. The final batch of records should set the isLast flag to true.

The maxBatchSize parameter passed to the nextBatch method corresponds to the Connector Batch Size that can be set on a system-wide bases in the Teiid Console (under System Properties in the buffer section..) The Connector Batch Size should typically be set in conjunction with the

Processor Batch Size from the same category. For more information on these parameters, see the Teiid Console User Guide.

3.4.3. Asynchronous Query Execution

In some scenarios, a connector needs to execute queries asynchronously and poll for results. In this case, your connector should use the asynchronous query execution mode instead of the synchronous query execution mode. The connector capabilities specify which will be used (only one can be supported at the same time).

The following diagram represents the typical sequence of events when executing a query asynchronously:

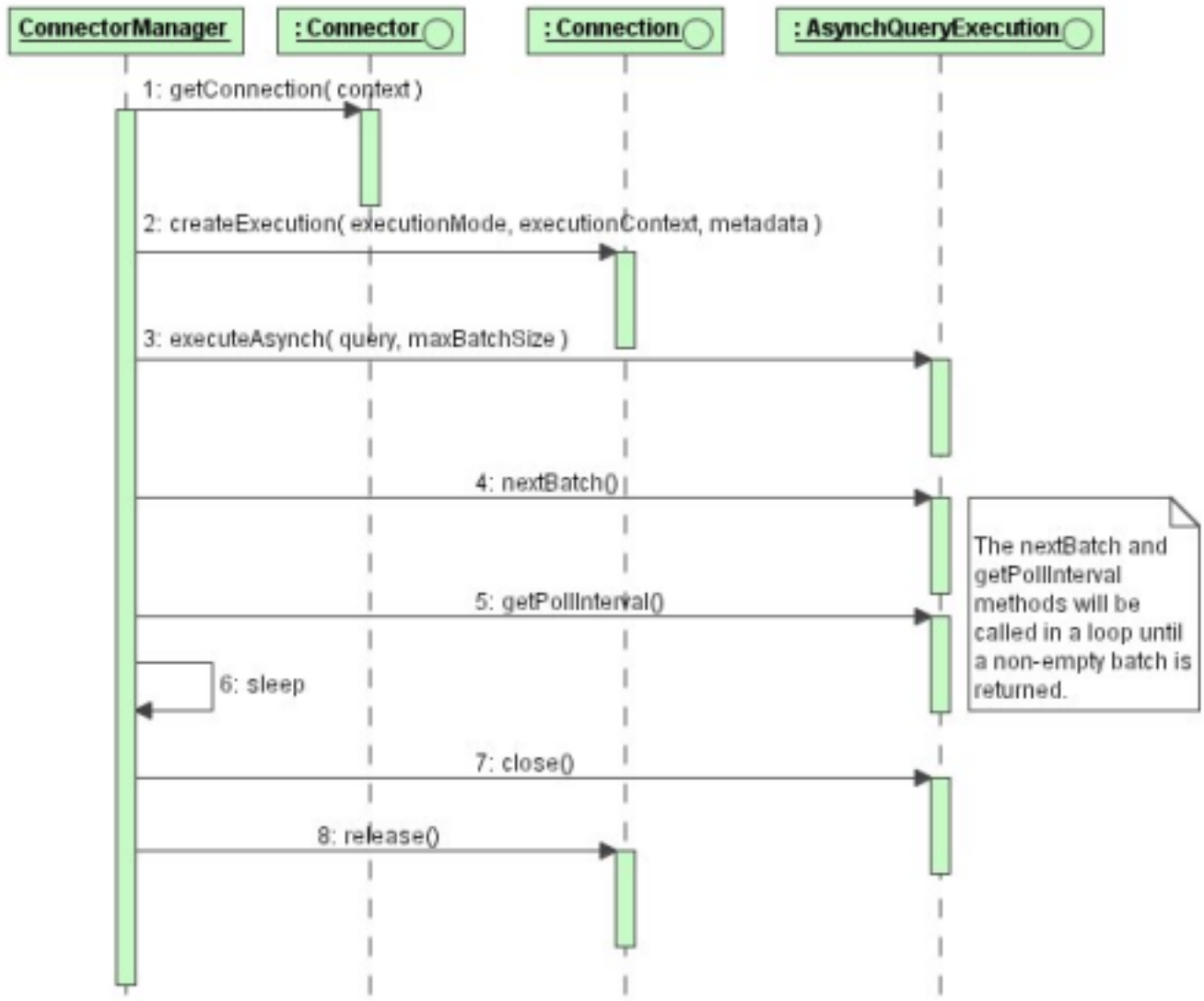

Figure 3.3. Async Query Execution Sequence Diagram

While the command is being executed, the connector retrieves results in batches via the BatchedExecution interface. The AsynchQueryExecution interface works similarly to the

SynchQueryExecution interface with one important difference. If the nextBatch method returns an empty batch with the isLast flag set to false, then the Connector Manager interprets this as no data being available. In this case, the Connector Manager will wait for the poll interval before asking again for a batch.

The nextBatch() is not expected to sleep or otherwise block for results as this would tie up a Connector Manager thread which could be doing other work. Instead, the Connector Manager is designed to avoid tying up worker threads while waiting for the poll interval, so the connector is expected to return from the nextBatch() method as quickly as possible.

3.4.4. Update Execution

Insert, update, and delete commands correspond to the INSERT, UPDATE, and DELETE commands in SQL. They are used to insert a single row into a data source, update one or more rows in a data source, or delete one or more rows in a data source. Each of these commands returns a count specifying the number of rows updated in the physical source in response to the command.

The following diagram represents the typical sequence of events when executing an insert, update, or delete:

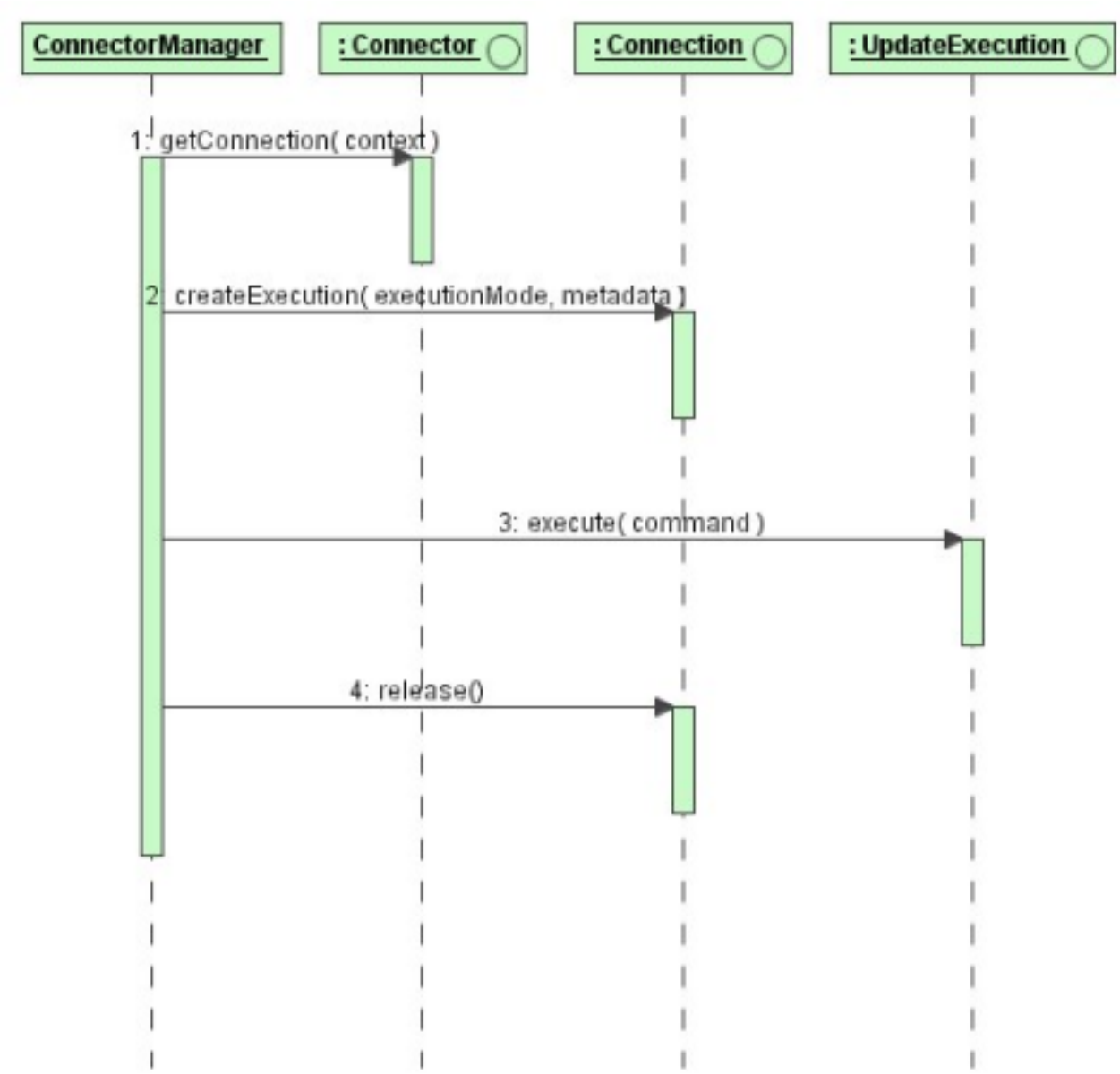

Figure 3.4. Update Query Execution Sequence Diagram

With an update execution, the execute method is the only call in the execution. No subsequent interaction will take place.

3.4.5. Batched Update / Bulk Insert Execution

Batched update and bulk insert execution are very similar to update execution except multiple commands are passed to the connector in a single method call. In the case of a bulk insert, a single template INSERT command is passed with a set of data rows that need to be inserted with the command. In the case of batched update, a series of arbitrary commands is sent and the batch must be executed together for efficiency.

3.4.6. Procedure Execution

Procedure commands correspond to the execution of a stored procedure or some other functional construct. A procedure takes zero or more input values and can return a result set and zero or more output values. Examples of procedure execution would be a stored procedure in a relational database or a call to a web service.

The following diagram represents the typical sequence of events when executing a procedure:

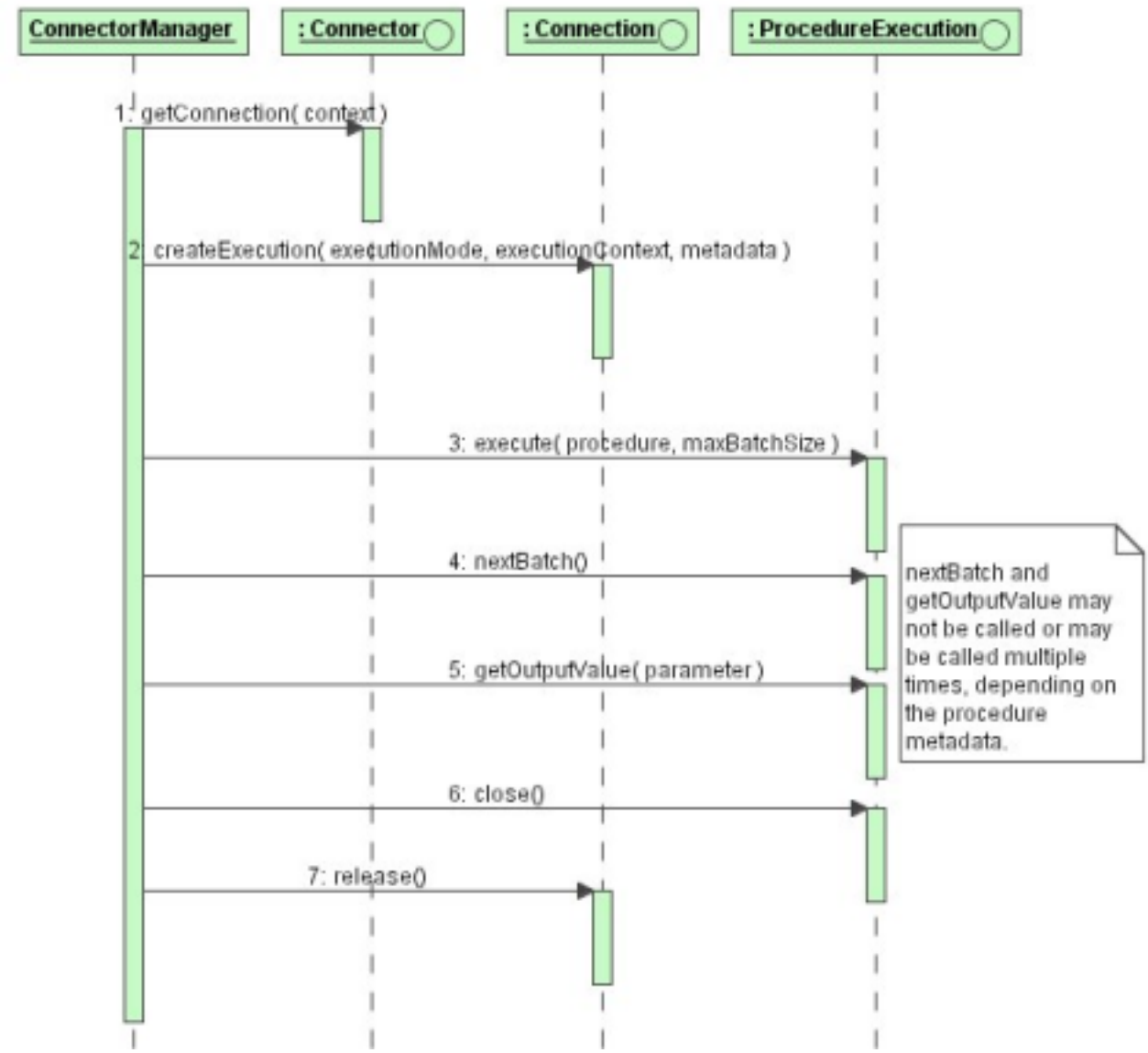

Figure 3.5. Procedure Query Execution Sequence Diagram

If a result set is expected when a procedure is executed, all rows from it will be retrieved via the BatchedExecution interface first. Then, if any output values are expected, they will be retrieved via the getOutputValue() method, which will be called once for each expected output value.

3.4.7. Command Completion

All normal command executions end with the calling of close() on the Execution object. Your implementation of this method should do the appropriate clean-up work for all state in the Execution object.

3.4.8. Command Cancellation

Commands submitted to Teiid may be aborted in several scenarios:

- Client cancellation via the JDBC API (or other client APIs)
- Administrative cancellation via the Teiid Console or Admin API
- Clean-up during session termination
- Clean-up if a query fails during processing

In these cases, if the command being executed on the connector has not yet finished, Teiid will call the cancel() method on the execution interface in a separate thread from the thread that may be blocked calling the execute() method.

Your connector implementation may choose to do nothing in response to this cancellation message. In this instance, Teiid will call close() on the execution object after current synchronous processing has completed. Implementing the cancel() method allows for faster termination of queries being processed and may allow the underlying data source to terminate its operations faster as well.

Command Language

4.1. Language Interfaces

Teiid sends commands to your connector in object form. The interfaces for these objects are all defined in the com.metamatrix.data.language package. These interfaces can be combined to represent any possible command that Teiid may send to the connector. However, it is possible to notify the Teiid Server that your connector can only accept certain kinds of commands via the ConnectorCapabilities class. See the section on using *Connector Capabilities* for more information.

The language interfaces all extend from the main interface, ILanguageObject. They are composed in a tree where each node is a language object that has zero or more child language objects of types that are dependent on the current node.

All commands sent to your connector are in the form of these language trees, where the root of the tree is a subclass of ICommand. ICommand has several sub-interfaces, namely: IQuery, IInsert, IUpdate, IDelete, and IProcedure. These represent the query in SQL form. Important components of these commands are expressions, criteria, and joins, which are examined in closer detail below.

4.1.1. Expressions

An expression represents a single value in context, although in some cases that value may change as the query is evaluated. For example, a literal value, such as “5” represents an integer value. An element such as “EmployeeName” represents a column in a data source and may take on many scalar values while the command is being evaluated.

The following diagram shows the IExpression interface and all sub-interfaces in the language objects.

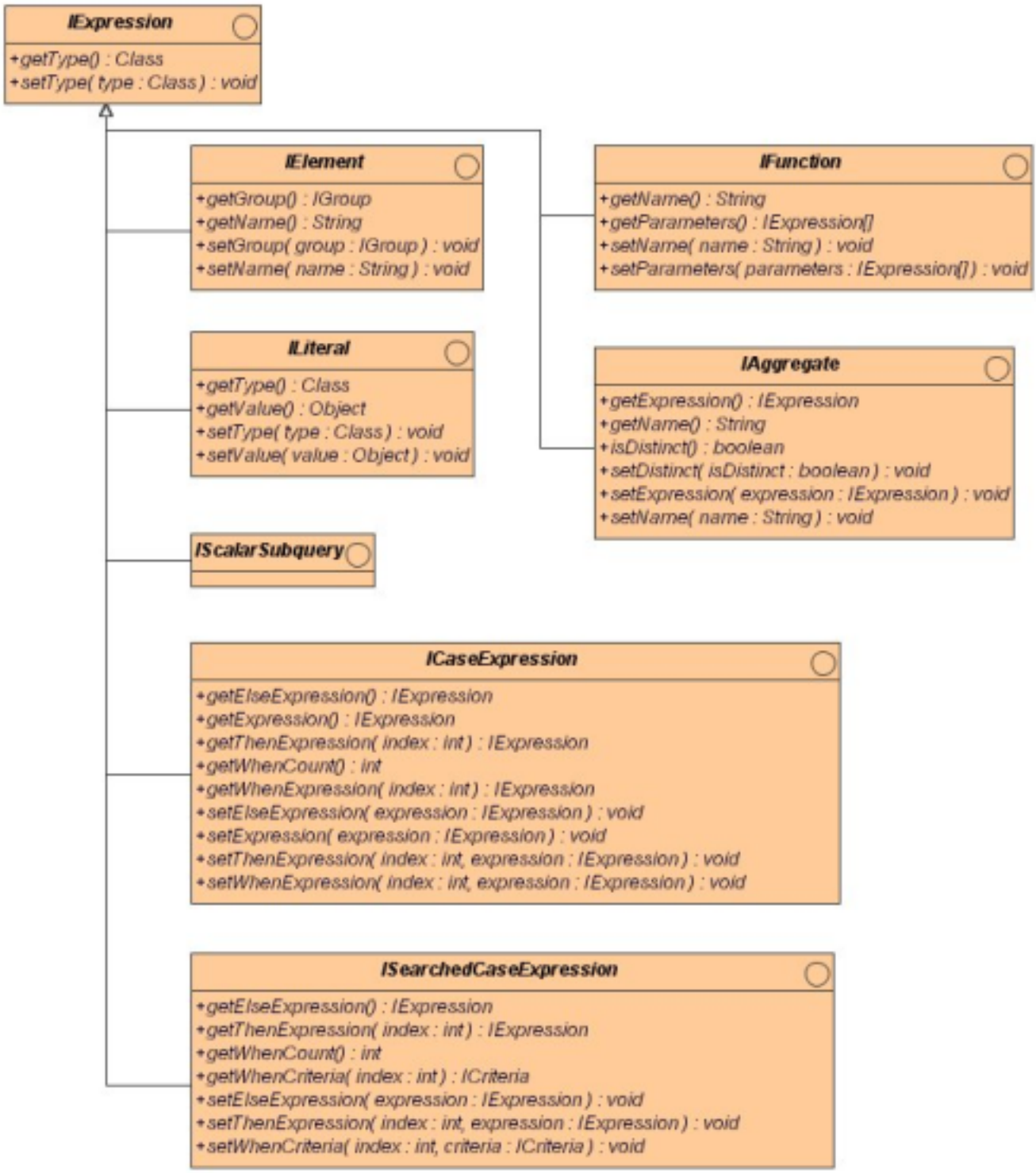

Figure 4.1. Execution Interfaces Class Diagram

These interfaces are explained in greater detail here:

- *IExpression* – base expression interface

- *IElement* – represents an element in the data source
- *ILiteral* – represents a literal scalar value
- *IFunction* – represents a scalar function with parameters that are also IExpressions
- *IAggregate* – represents an aggregate function which holds a single expression
- *IScalarSubquery* – represents a subquery that returns a single value
- *ICaseExpression* – represents a CASE expression. The CASE expression evaluates an expression, then compares with the values in the WHEN clauses to determine the THEN clause to evaluate.
- *ISearchedCaseExpression* – represents a searched CASE expression. The searched CASE expression evaluates the criteria in WHEN clauses till one evaluates to TRUE, then evaluates the associated THEN clause.

4.1.2. Criteria

A criteria is a combination of expressions and operators that evaluates to true or false. Criteria are most commonly used in the FROM or HAVING clauses. The following diagram shows the criteria interfaces present in the language objects.

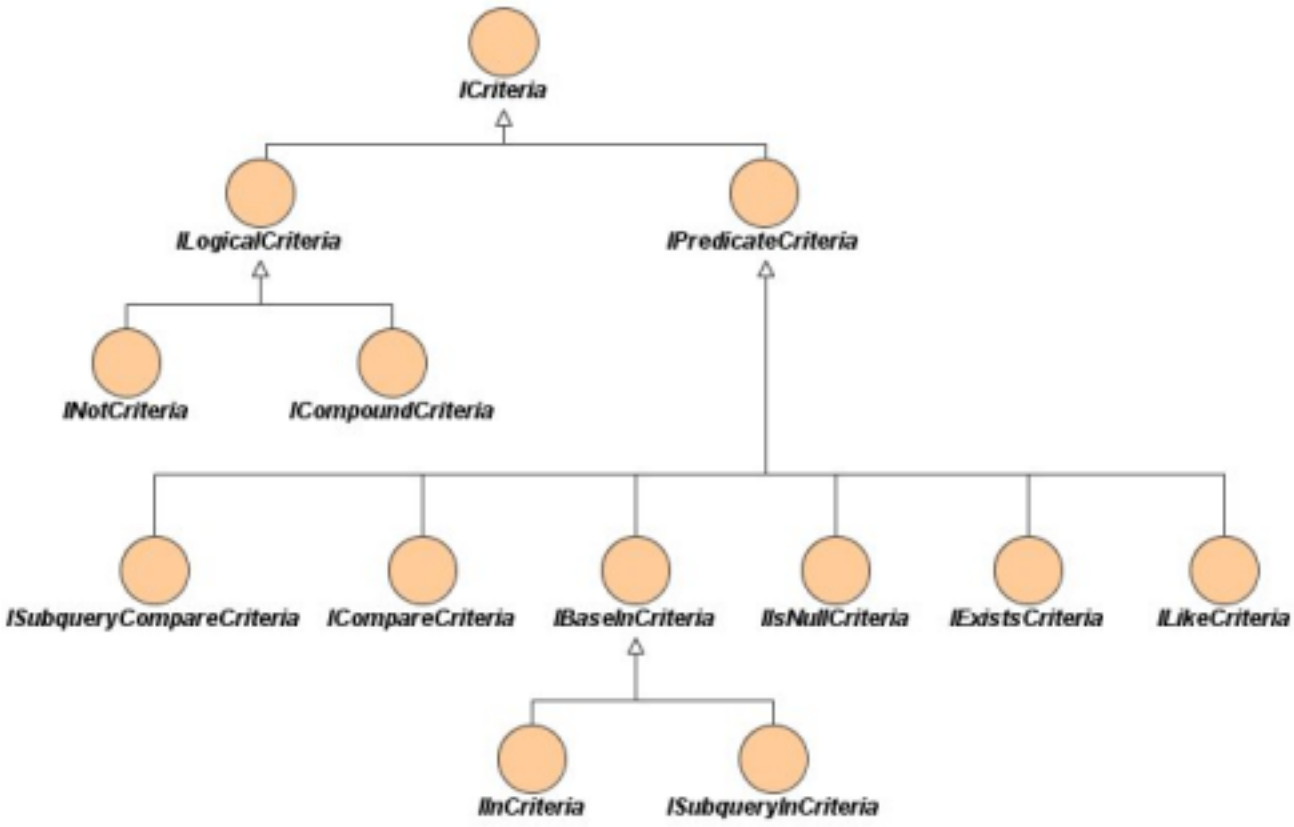

Figure 4.2. Criteria Interfaces Class Diagram

These interfaces are described in greater detail here:

- *ICriteria* – the base criteria interface
- *ILogicalCriteria* – used to logically combine other criteria
- *INotCriteria* – used to NOT another criteria
- *ICompoundCriteria* – used to combine other criteria via AND or OR
- *IPredicateCriteria* – a predicate that evaluates to true or false
- *ISubuqeryCompareCriteria* – represents a comparison criteria with a subquery including a quantifier such as *SOME* or *ALL*
- *ICompareCriteria* – represents a comparison criteria with =, >, <, etc
- *IBaseInCriteria* – base class for an *IN* criteria
- *IInCriteria* – represents an *IN* criteria that has a set of expressions for values
- *ISubqueryInCriteria* – represents an *IN* criteria that uses a subquery to produce the value set
- *IIsNullCriteria* – represents an *IS NULL* criteria
- *IExistsCriteria* – represents an *EXISTS* criteria that determines whether a subquery will return any values
- *ILikeCriteria* – represents a *LIKE* criteria that compares string values

4.1.3. Joins

The FROM clause contains a list of IFromItems. Each IFomItem can either represent a group or a join between two other IFromItems. This allows joins to be composed into a join tree.

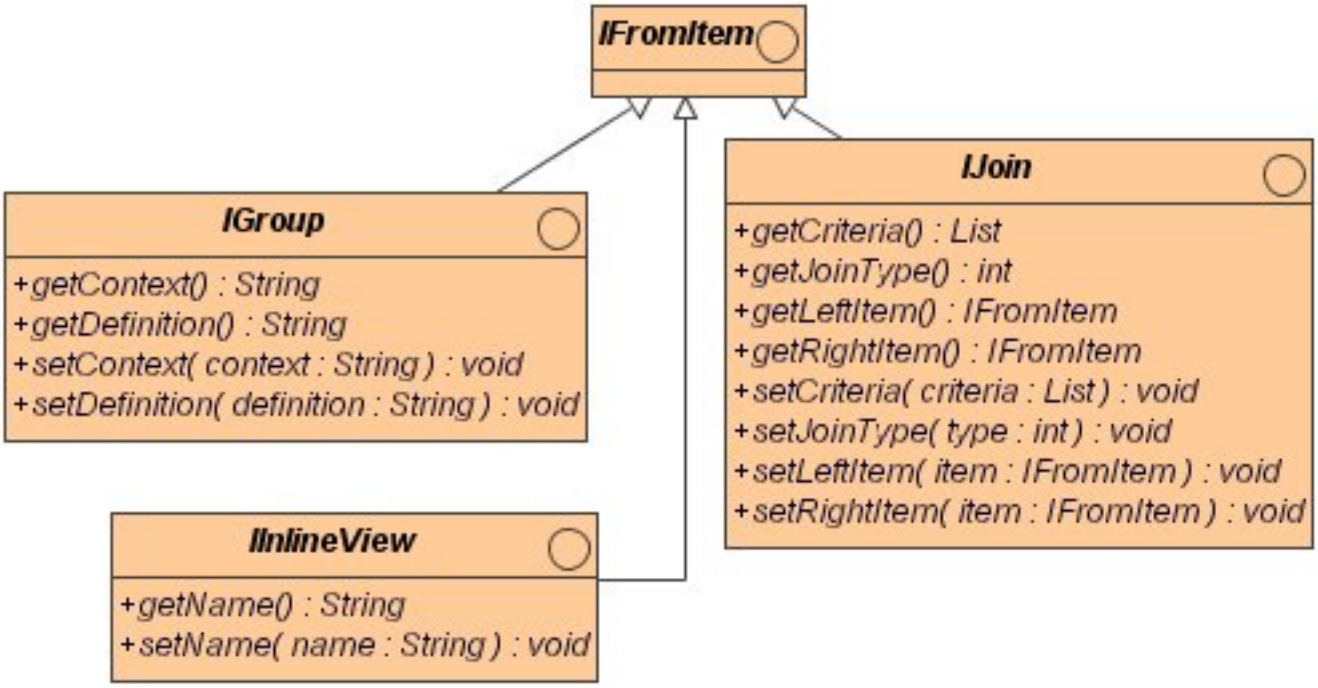

Figure 4.3. IJoin Interface Class Diagram

The `IGroup` represents a single group, which must be a leaf of the join tree. The `IJoin` has a left and right `IFromItem` and information on the join between the items.

4.1.4. IQuery Structure

The following diagram shows the structure of an `IQuery` command.

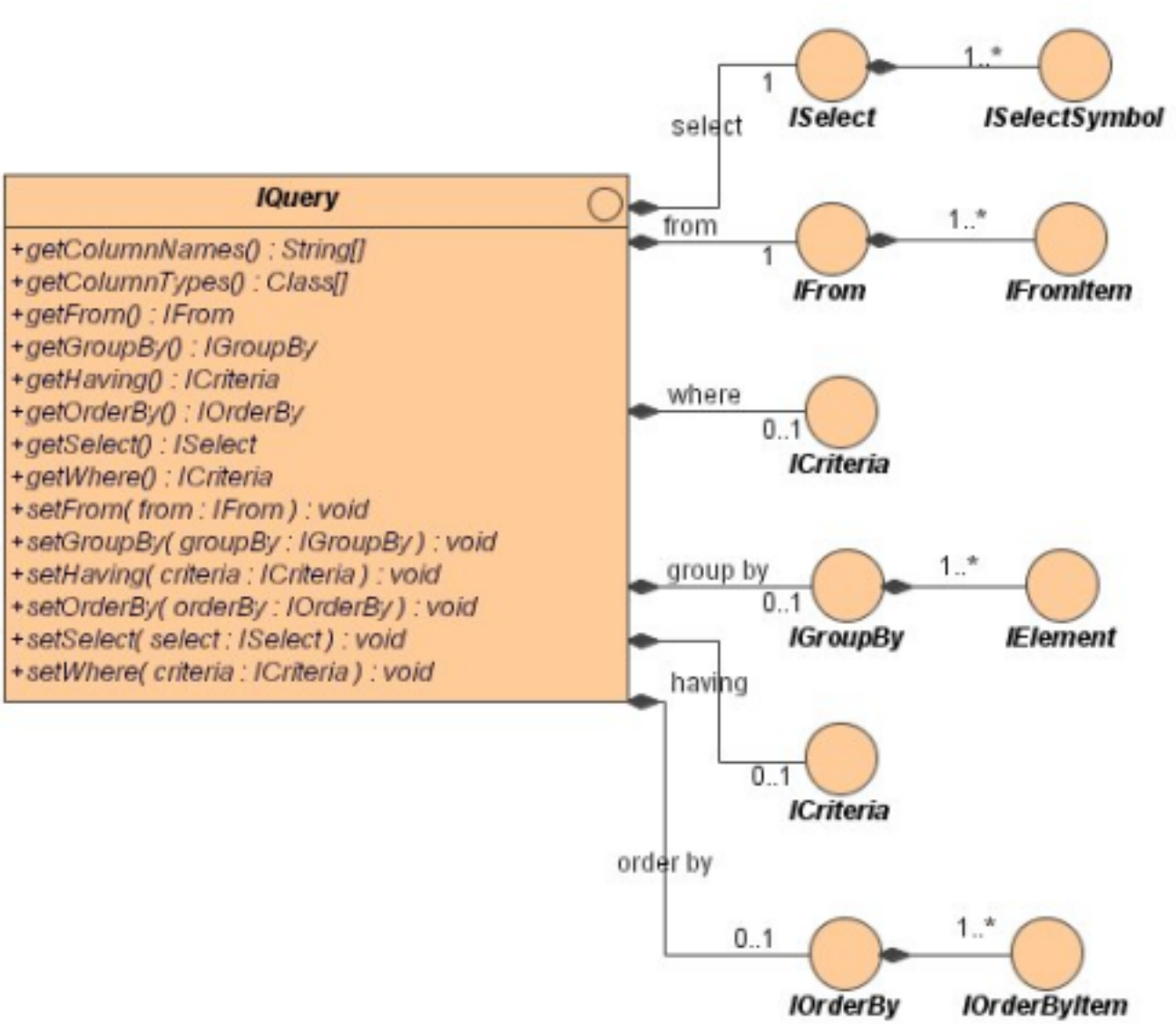

Figure 4.4. IQuery Interface Class Diagram

Each IQuery will have an ISelect describing the expressions (typically elements) being selected and an IFrom specifying the group or groups being selected from, along with any join information. The IQuery may optionally also supply an ICriteria (representing a SQL WHERE clause), an IGroupBy (representing a SQL GROUP BY clause), and an ICriteria (representing a SQL HAVING clause).

4.1.5. IUnion Structure

The following diagram shows the structure of an IUnion command.

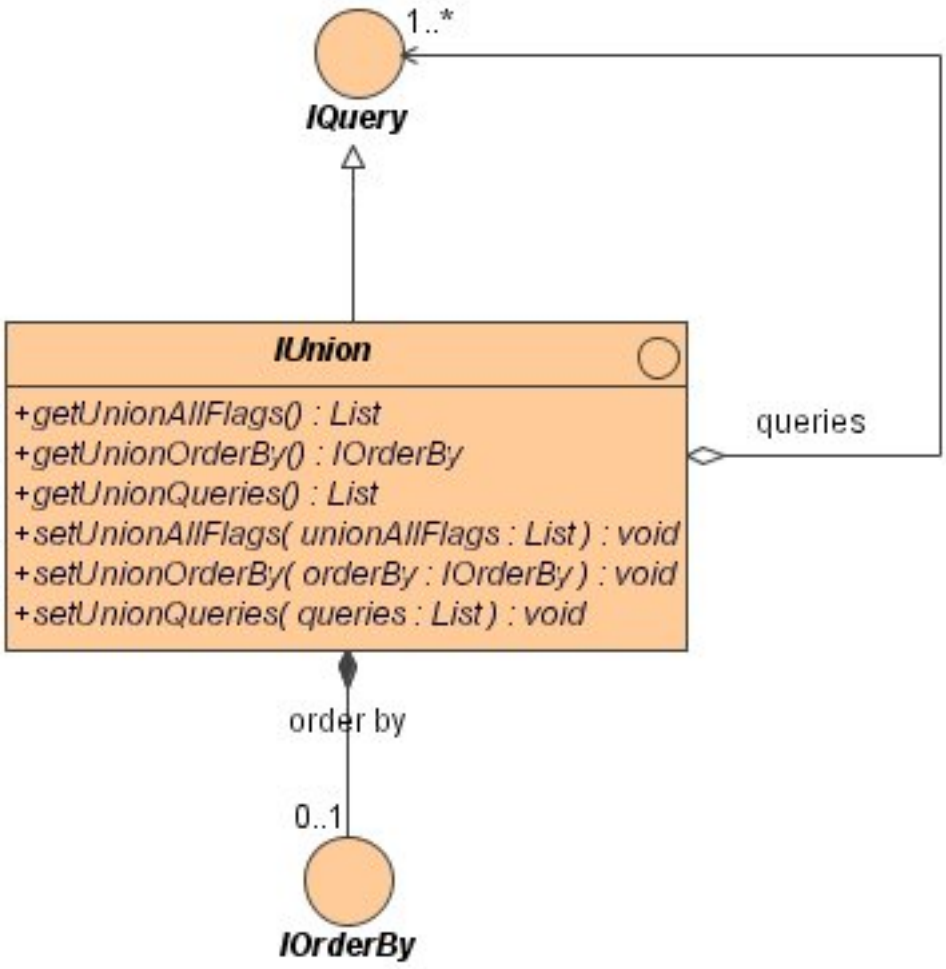

Figure 4.5. IUnion Interface Class Diagram

IUnion extends from IQuery and allows for one or more additional IQuery objects to be attached as a UNION query. For each additional IQuery, there is a Boolean “ALL” flag that must be set. The ORDER BY clause for the UNION as a whole can also be set on the IUnion object.

4.1.6. IInsert Structure

The following diagram shows the structure of an IInsert command.

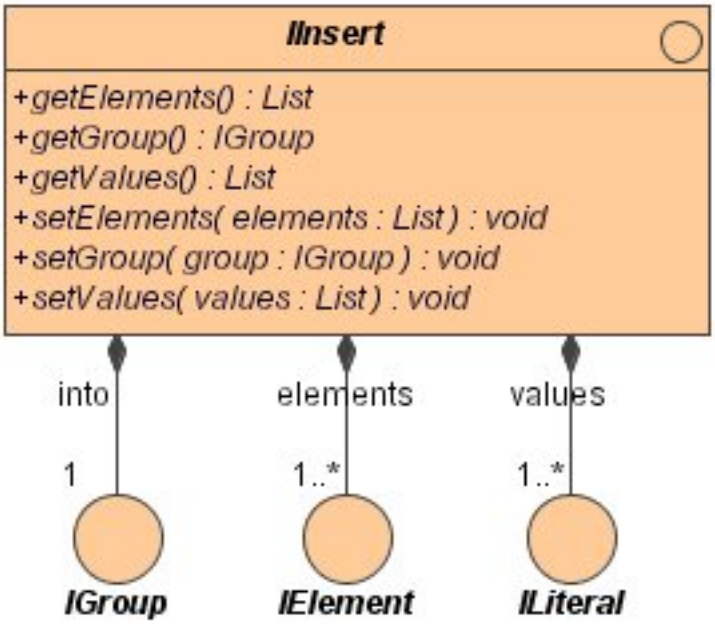

Figure 4.6. IInsert Interface Class Diagram

Each IInsert will have a single IGroup specifying the group being inserted into. It will also have two matched lists – one a list of IElement specifying the columns of the IGroup that are being inserted into and one a list of ILiteral specifying the values that will be inserted into each matching IElement.

4.1.7. IUpdate Structure

The following diagram shows the structure of an IUpdate command.

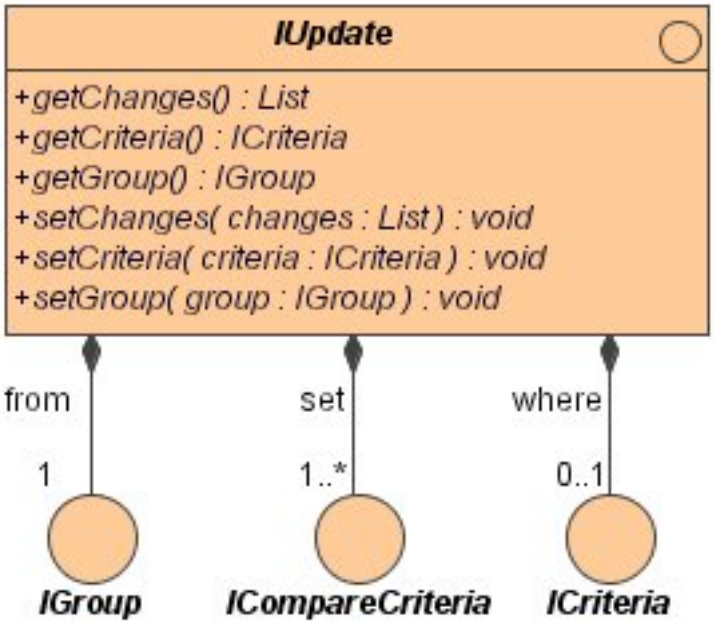

Figure 4.7. IUpdate Interface Class Diagram

Each IUpdate will have a single IGroup specifying the group being updated. The list of ICompareCriteria are used to specify each element that is being modified. Each compare criteria will be of the form “element = literal”. The IUpdate may optionally provide a criterion specifying which rows should be updated.

4.1.8. IDelete Structure

The following diagram shows the structure of an IDelete command.

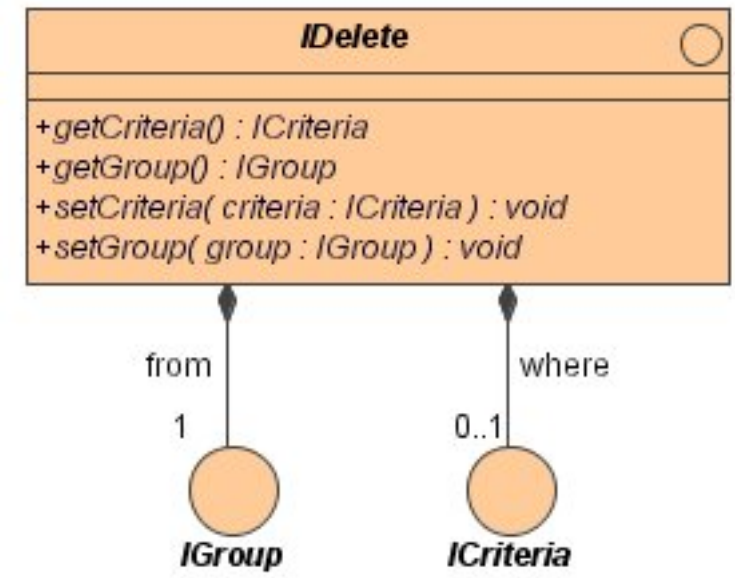

Figure 4.8. IDelete Interface Class Diagram

Each IDelete will have a single IGroup specifying the group being deleted from. It may also optionally have a criteria specifying which rows should be deleted.

4.1.9. IProcedure Structure

The following diagram shows the structure of an IProcedure command.

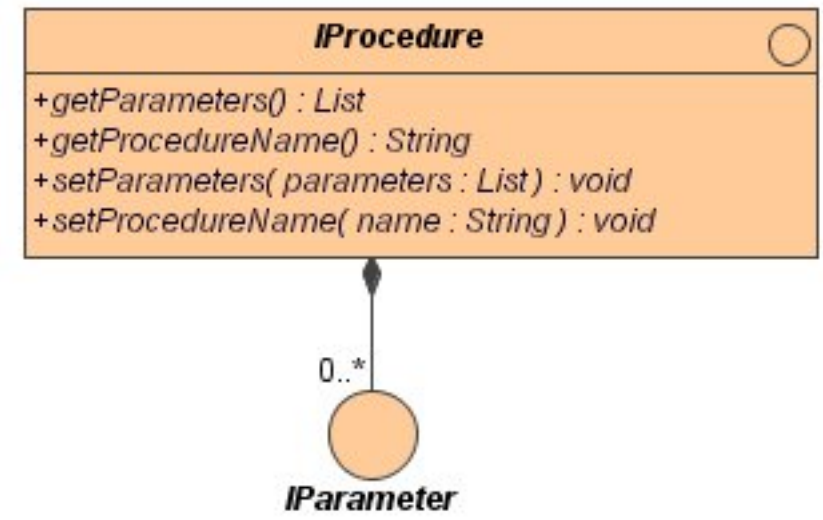

Figure 4.9. IProcedure Interface Class Diagram

Each IProcedure has zero or more IParameter objects. The IParameter objects describe the input parameters, the output result set, and the output parameters.

4.1.10. IBulkInsert Structure

The following diagram shows the structure of an IBulkInsert command.

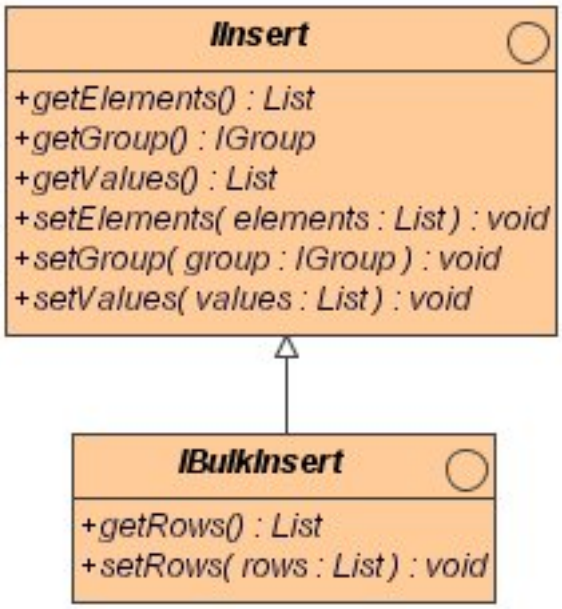

Figure 4.10. IBulkInsert Interface Class Diagram

Each IBulkInsert extends an IInsert but provides a List of Lists of values that need to be placed into the VALUES list of the INSERT.

4.1.11. IBatchedUpdate Structure

The following diagram shows the structure of an IBatchedUpdate command.

Figure 4.11. IBatchedUpdate Interface Class Diagram

Each IBatchedUpdate has a list of ICommand objects that compose the batch.

4.2. Language Utilities

This section covers utilities available when using, creating, and manipulating the language interfaces.

4.2.1. Data Types

The Connector API contains an interface TypeFacility that defines data types and provides value translation facilities.

Figure 4.12. IBatchedUpdate Interface Class Diagram

The table below lists the role of each class in the framework.

Table 4.1. Types Facility Classes

Class	Type	Description
ConnectorEnvironment	Interface	This interface (provided by Teiid) is a factory to obtain the TypeFacility instance for the connector using the getTypeFacility() method.
TypeFacility	Interface	This interface has two methods that support data type transformation. Generally,transformations exist for all valid implicit and explicit data type transformations in the Teiid query engine.
TypeFacility.RUNTIME_TYPES	Interface	This is an inner interface of TypeFacility that defines constants for all Teiid runtime data types. All IExpression instances define a data type based on this set of types. These constants are often needed in understanding or creating language interfaces.

4.2.2. Language Manipulation

In connectors that support a fuller set of capabilities (those that generally are translating to a language of comparable to SQL), there is often a need to manipulate or create language interfaces to move closer to the syntax of choice. Some utilities are provided for this purpose:

Similar to the TypeFacility, you can use the ConnectorEnvironment to get a reference to the ILanguageFactory instance for your connector. This interface is a factory that can be used to create new instances of all the concrete language interface objects.

Some helpful utilities for working with ICriteria objects are provided in the LanguageUtil class. This class has methods to combine ICriteria with AND or to break an ICriteria apart based on AND operators. These utilities are helpful for breaking apart a criteria into individual filters that your connector can implement.

4.3. Runtime Metadata

Teiid uses a library of metadata, known as “runtime metadata” for each virtual database that is deployed in the Teiid Server. The runtime metadata is a subset of metadata as defined by models in the Teiid models that compose the virtual database.

Connectors can access runtime metadata by using the interfaces defined in com.metamatrix.data.metadata.runtime. This class defines interfaces representing a MetadataID, a MetadataObject, and ways to navigate those IDs and objects.

4.3.1. Language Objects

One language interface, IMetadataReference describes whether a language object has a reference to a MetadataID. The following interfaces extend IMetadataReference:

- IElement
- IGroup
- IProcedure
- IParameter

Once a MetadataID has been obtained, it is possible to use the RuntimeMetadata interface to discover metadata about that ID or to find other related IDs or objects.

4.3.2. Access to Runtime Metadata

The following interfaces are defined in the runtime metadata package:

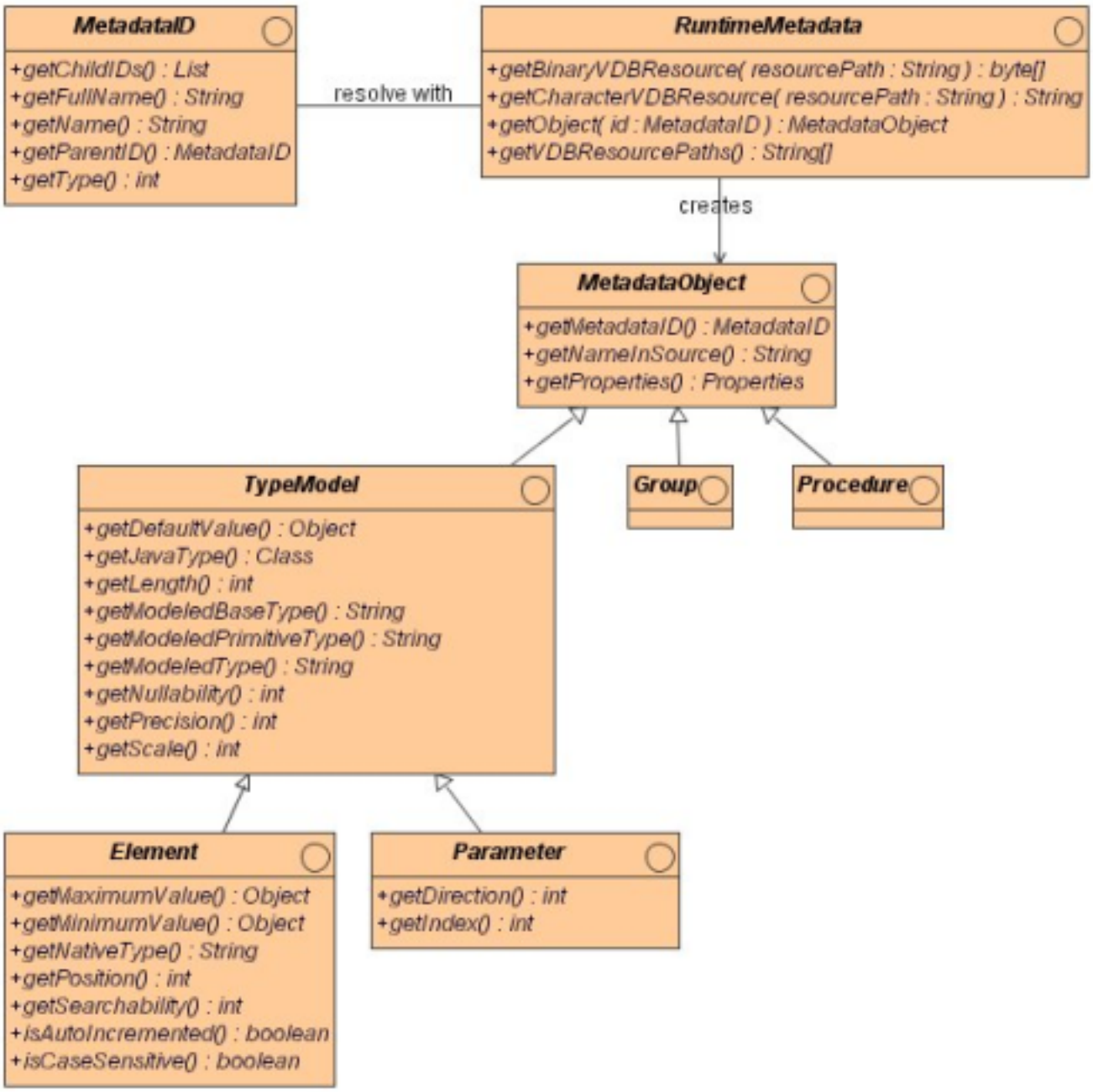

Figure 4.13. Runtime MetaData Class Diagram

As mentioned in the previous section, a MetadataID is obtained from one of the language objects. That MetadataID can then be used directly to obtain information about the ID, such as the full name or short name.

The RuntimeMetadata interface can be obtained from the ConnectorEnvironment. It provides the ability to look up MetadataObjects based on MetadataIDs. There are several kinds of MetadataObjects and they can be used to find more information about the object in runtime metadata.

Currently, only a subset of the most commonly used runtime metadata is available through these interfaces. In the future, more complete information will be available.

Obtaining MetadataObject Properties Example

The process of getting an element's properties is needed for most connector development For example to get the NameInSource property or all extension properties:

```
IMetaDataReference ref = ... //An IGroup in this example
RuntimeMetadata rm = ... //Obtained from the ConnectorEnvironemnt

MetaDataObject group = rm.getObject(ref.getMetadataID());
String contextName = group.getNameInSource();

//The props will contain extension properties
Properties props = group.getProperties();
```

4.4. Language Visitors

4.4.1. Framework

The Connector API provides a language visitor framework in the com.metamatrix.data.visitor.framework package. The framework provides utilities useful in navigating and extracting information from trees of language objects. This diagram describes the relationships of the various classes in the framework:

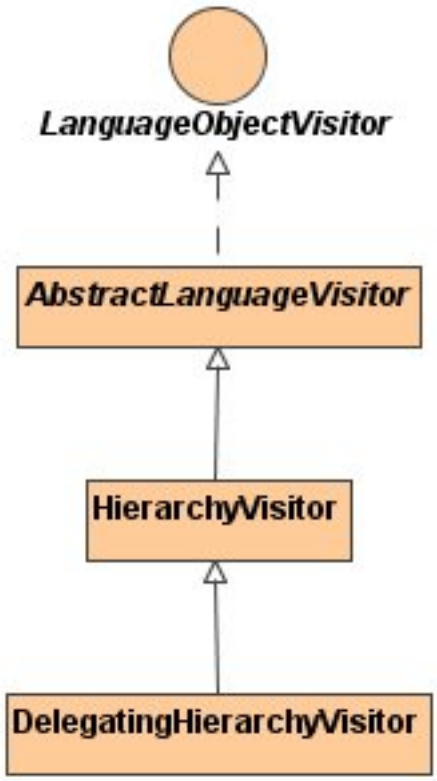

Figure 4.14. LanguageObjectVisitor Class Diagram

The visitor framework is a variant of the Visitor design pattern, which is documented in several popular design pattern references. The visitor pattern encompasses two primary operations: traversing the nodes of a graph (also known as iteration) and performing some action at each node of the graph. In this case, the nodes are language interface objects and the graph is really a tree rooted at some node. The provided framework allows for customization of both aspects of visiting.

The base LanguageObjectVisitor class defines the visit methods for all leaf language interfaces that can exist in the tree. The LanguageObject interface defines an acceptVisitor() method – this method will call back on the visit method of the visitor to complete the contract. A base class with empty visit methods is provided as AbstractLanguageVisitor. The AbstractLanguageVisitor is just a visitor shell – it performs no actions when visiting nodes and does not provide any iteration.

The HierarchyVisitor provides the basic code for walking a language object tree. The HierarchyVisitor performs no action as it walks the tree – it just encapsulates the knowledge of how to walk it. If your connector wants to provide a custom iteration that walks the objects in a special order (to exclude nodes, include nodes multiple times, conditionally include nodes, etc) then you must either extend HierarchyVisitor or build your own iteration visitor. In general, that is not necessary.

The DelegatingHierarchyVisitor is a special subclass of the HierarchyVisitor that provides the ability to perform a different visitor's processing before and after iteration. This allows users of this class to implement either pre- or post-order processing based on the HierarchyVisitor. Two

helper methods are provided on DelegatingHierarchyVisitor to aid in executing pre- and post-order visitors.

4.4.2. Provided Visitors

The SQLStringVisitor is a special visitor that can traverse a tree of language interfaces and output the equivalent Teiid SQL. This visitor can be used to print language objects for debugging and logging. The SQLStringVisitor does not use the HierarchyVisitor described in the last section; it provides both iteration and processing type functionality in a single custom visitor.

The CollectorVisitor is a handy utility to collect all language objects of a certain type in a tree. Some additional helper methods exist to do common tasks such as retrieving all elements in a tree, retrieving all groups in a tree, and so on.

4.4.3. Writing a Visitor

Writing your own visitor can be quite easy if you use the provided facilities. If the normal method of iterating the language tree is sufficient, then just follow these steps:

Create a subclass of AbstractLanguageVisitor. Override any visit methods needed for your processing. For instance, if you wanted to count the number of elements in the tree, you need only override the visit(IElement) method. Collect any state in local variables and provide accessor methods for that state.

Decide whether to use pre-order or post-order iteration. In many cases, it doesn't matter, so if you're not sure, use pre-order processing.

Write code to execute your visitor using the utility methods on DelegatingHierarchyVisitor:

```
// Get object tree
LanguageObject objectTree = …

// Create your visitor initialize as necessary
MyVisitor visitor = new MyVisitor();

// Call the visitor using pre-order visitation
DelegatingHierarchyVisitor.preOrderVisit(visitor, objectTree);

// Retrieve state collected while visiting
int count = visitor.getCount();
```

Often it's useful to create a static method implementing this sequence of calls within your visitor.

4.5. Connector Capabilities

All connectors must return a ConnectorCapabilities class from the Connection.getCapabilities() method. This class is used by the Connector Manager to determine what kinds of commands the connector is capable of executing. A basic implementation of the ConnectorCapabilities interface is supplied at com.metamatrix.data.basic.BasicConnectorCapabilities. This capabilities class specifies that the connector only executes queries and does not support any capability. Teiid recommends that you extend this class and override the necessary methods to specify which capabilities your connector supports.

4.5.1. Capability Scope

The method ConnectorCapabilities.getScope() specifies the scope of a capabilities set. Currently, two scope modes are defined in ConnectorCapabilities.SCOPE: global and per user. Specifying the scope as global means that the capabilities are the same for all connections to this source. Specifying the scope as per user means that the capabilities are potentially different for each user, so capabilities cannot be cached between users.

The per user mode is significantly slower and usually not necessary, therefore Teiid recommends using the global mode if capabilities of a source are the same across all connections. The BasicConnectorCapabilities implementation specifies global scope.

4.5.2. Execution Modes

The method ConnectorCapabilities.supportsExecutionMode() is used by the Connector Manager to discover what kinds of commands the connector can support. Constants defining the available execution modes are specified in ConnectorCapabilities.EXECUTION_MODE. Your implementation of ConnectorCapabilities should return true from this method for each execution mode your connector supports.

The BasicConnectorCapabilities implementation specifies only that it supports the SYNCH_QUERY execution mode.

4.5.3. Capabilities

The following table lists the capabilities that can be specified in the ConnectorCapabilities class.

Table 4.2. Available Connector Capabilities

Capability	Requires	Description
SelectDistinct		Connector can support SELECT DISTINCT in queries.
Joins		Connector can support joins.
OuterJoins	Joins	Connector can support LEFT and RIGHT OUTER JOIN.
FullOuterJoins	Joins, OuterJoins	Connector can support FULL OUTER JOIN.

Capability	Requires	Description
AliasedGroup		Connector can support groups in the FROM clause that have an alias.
SelfJoins	Joins, AliasedGroups	Connector can support a self join between two aliased versions of the same group.
InlineViews	AliasedGroup	Connector can support a named subquery in the FROM clause.
Criteria		Connector can support WHERE and HAVING clauses.
RowLimit		Connector can support the limit portion of the limit clause
RowOffset		Connector can support the offset portion of the limit clause
AndCriteria	Criteria	Connector can support AND criteria in join conditions of the FROM clause, the WHERE clause, and the HAVING clause.
OrCriteria	Criteria	Connector can support the OR logical criteria.
NotCriteria	Criteria	Connector can support the NOT logical criteria.
BetweenCriteria	Criteria	Connector can support the BETWEEN predicate criteria.
CompareCriteria	Criteria	Connector can support comparison criteria such as “age > 10”.
CompareCriteriaEquals	Criteria, CompareCriteria	Connector can support comparison criteria with the operator “=”.
CompareCriteriaGreaterThan	Criteria, CompareCriteria	Connector can support comparison criteria with the operator “>”.
CompareCriteriaGreaterThanOrEqual	Criteria, CompareCriteria	Connector can support comparison criteria with the operator “>=”.
CompareCriteriaLessThan	Criteria, CompareCriteria	Connector can support comparison criteria with the operator “<”.
CompareCriteriaLessThanOrEqual	Criteria, CompareCriteria	Connector can support comparison criteria with the operator “<=”.
CompareCriteriaNotEquals	Criteria, CompareCriteria	Connector can support comparison criteria with the operator “<>”.
ExistsCriteria	Criteria	Connector can support EXISTS predicate criteria.
InCriteria	Criteria	Connector can support IN predicate criteria.

Capability	Requires	Description
InCriteriaSubquery	Criteria, InCriteria	Connector can support IN predicate criteria where values are supplied by a subquery.
IsNullCriteria	Criteria	Connector can support IS NULL predicate criteria.
LikeCriteria	Criteria	Connector can support LIKE criteria.
LikeCriteriaEscapeCharacter	Criteria, LikeCriteria	Connector can support LIKE criteria with an ESCAPE character clause.
QuantifiedCompareCriteria	Criteria, CompareCriteria	Connector can support a quantified comparison criteria with a subquery on the right side.
QuantifiedCompareCriteriaAll	Criteria, CompareCriteria, QuantifiedCompareCriteria	Connector can support a quantified comparison criteria using the ALL quantifier.
QuantifiedCompareCriteriaSome	Criteria, CompareCriteria, QuantifiedCompareCriteria	Connector can support a quantified comparison criteria using the SOME or ANY quantifier.
OrderBy		Connector can support the ORDER BY clause in queries.
Aggregates		Connector can support GROUP BY and HAVING clauses in queries.
AggregatesAvg	Aggregates	Connector can support the AVG aggregate function.
AggregatesCount	Aggregates	Connector can support the COUNT aggregate function.
AggregatesCountStar	Aggregates, AggregatesCount	Connector can support the COUNT(*) aggregate function.
AggregatesDistinct	Aggregates	Connector can support the keyword DISTINCT inside an aggregate function. This keyword indicates that duplicate values within a group of rows will be ignored.
AggregatesMax	Aggregates	Connector can support the MAX aggregate function.
AggregatesMin	Aggregates	Connector can support the MIN aggregate function.
AggregatesSum	Aggregates	Connector can support the SUM aggregate function.
ScalarFunctions		Connector can support scalar functions wherever expressions are accepted.

Capability	Requires	Description
CaseExpressions		Connector can support “unsearched” CASE expressions anywhere that expressions are accepted.
SearchedCaseExpressions		Connector can support “searched” CASE expressions anywhere that expressions are accepted.
ScalarSubqueries		Connector can support the use of a subquery in a scalar context (wherever an expression is valid).
CorrelatedSubqueries	ScalarSubqueries or QuantifiedCompareCriteria or ExistsCriteria or InCriteriaSubquery	Connector can support a correlated subquery that refers back to an element in the outer query.
SelectLiterals		Connector can support literals in the SELECT clause
Unions		Connector support UNIONs
Intersect		Connector supports INTERSECT
Except		Connector supports Except
SetQueryOrderBy	Unions, Intersect, or Except	Connector supports set queries with an ORDER BY
FunctionsInGroupBy	ScalarFunctions, Aggregates	Connector supports functions in the GROUP BY list
FunctionsInGroupBy	ScalarFunctions, Aggregates	Connector supports functions in the GROUP BY list

4.5.4. Command Form

The method ConnectorCapabilities.useAnsiJoin() should return true if the Connector prefers the use of ANSI style join structure for INNER and CROSS joins that are pushed down.

The method ConnectorCapabilities.requiresCriteria() should return true if the Connector requires criteria for any Query, Update, or Delete. This is a replacement for the model support property "Where All".

4.5.5. Scalar Functions

The method ConnectorCapabilities.getSupportedFunctions() can be used to specify which scalar functions the connector supports. The set of possible functions is based on the set of functions supported by Teiid. This set can be found in the Query Support Booklet documentation. If the connector states that it supports a function, it must support all type combinations and overloaded forms of that function.

There are five operators that can also be specified in the supported function list: +, -, *, /, and ||.

4.5.6. Physical Limits

The method ConnectorCapabilities.getMaxInCriteriaSize() can be used to specify the maximum number of values that can be passed in an IN criteria. This is an important constraint as an IN criteria is frequently used to pass criteria between one source and another using a dependent join.

The method ConnectorCapabilities.getMaxFromGroups() can be used to specify the maximum number of FROM Clause groups that can used in a join. -1 indicates there is no limit.

Using the Connector Development Kit

5.1. Overview

The Connector Developer Kit (CDK) is a set of programmatic and command line utilities for testing connectors. The programmatic components of the CDK are useful for unit testing your connector and the command line utilities is useful for integration testing and regression testing (due to scripting abilities).

This chapter covers usage of both aspects of the CDK. For more detailed information about the CDK programmatic utilities, please see the Connector API Javadoc, which include the CDK Javadoc.

5.2. Programmatic Utilities

All components provided by the CDK are in the package com.metamatrix.cdk.api.

5.2.1. Language Translation

Commands are sent to the Connector API in terms of the language interfaces discussed earlier in this guide. Typically, a connector must write logic to read and sometimes manipulate these objects. The CDK language translation utilities can be used to write unit tests for translation code or command execution.

The utilities are provided in the class TranslationUtility. This class has the following methods:

Table 5.1. Language Translation

Method Name	Description
TranslationUtility(String vdbFile)	Constructor – takes the path to a file which is a valid metadata archive created by the Teiid Designer. These files have the suffix ".vdb".
createRuntimeMetadata()	Creates an instance of RuntimeMetadata that can be used to test code that uses runtime metadata when translating or executing commands.
parseCommand(String sql)	Take a single-source command and return an ICommand that can be used to test translation or execution of commands.

5.2.2. Command Execution

The primary purpose of a Connector is to execute commands against an information source. The query execution utilities allow you to test the execution of commands programmatically. This

utility does not run the Teiid query engine or the connector manager although does simulate what happens when those components use a Connector to execute a command.

The command execution utilities are provided in the class ConnectorHost. This class has the following methods:

Table 5.2. Command Execution

Method Name	Description
ConnectorHost	Constructor – takes a Connector instance, a set of connector property values, and the path to a VDB archive file
setBatchSize	Sets the batch size to use when executing commands.
setSecurityContext	Sets the security context values currently being used to execute commands. This method may be called multiple times during the use of a single instance of ConnectorHost to change the current context.
getConnectorEnvironmentProperties	Helper method to retrieve the properties passed to the ConnectorHost constructor.
executeCommand	Execute a command and return the results using this connector.
executeBatchedUpdates	Execute a set of commands as a batched update.
getCommand	Use the host metadata to get the ICommand for a SQL string.

Here is some example code showing how to use ConnectorHost to test a connector:

```
// Prepare state for testing
MyConnector connector = new MyConnector();
Properties props = new Properties();
props.setProperty(“user”, “myuser”);
props.setProperty(“password”, “mypassword”);
String vdbFile = “c:/mymetadata.vdb”;

// Create host
ConnectorHost host = new ConnectorHost(connector, props, vdbFile);

// Execute query
List results = host.executeCommand(“SELECT col FROM group WHERE col = 5”);

// Compare actual results to expected results
// . . .
```

The executeCommand() method will return results as a List of rows. Each row is itself a List of objects in column order. So, each row should have the same number of items corresponding to the columns in the SELECT clause of the query. In the case of an INSERT, UPDATE, or DELETE, a single “row” will be returned with a single column that contains the update count.

5.3. Connector Environment

Many parts of the Connector API require use of the Connector Environment. The EnvironmentUtility can be used to obtain and control a Connector Environment instance.

Table 5.3. Command Execution

Method Name	Description
createSecurityContext	Creates a securityContext instance.
createStdoutLogger	Creates an instance of ConnectorLogger that prints log messages to system.out( )
createEnvironment	Creates an instance of connectorEnvironment for use in your testing environment.
createExecutionContext	Creates an ExecutionContext instance.

In addition, some implementations of ConnectorLogger are provided which can be used as needed to build a custom logger for testing. BaseLogger is a base logger class that can be extended to create your own ConnectorLogger implementation. SysLogger is a utility implementation that logs to System.out.

5.4. Command Line Tester

5.4.1. Using the Command Line Tester

The command line tester is available in the mmtools kit along with the other Teiid products in the tools directory. The tester can be executed in interactive mode by running

```
<unzipped folder>S\cdk\cdk.bat
```

Typing “help” in the command line tester provides a list of all available options. These options are listed here with some additional detail:

Table 5.4. Connector Lifecycle

Option	Arguments	Description
Load Archive	PathToArchiveFileName	

Option	Arguments	Description
		Load the Connector archive file, which loads the Connector type definition file and all the extension modules into the CDK shell.
Load	ConnectorClass vdbFile	Load a connector by specifying the connector class name and the VDB metadata archive file
LoadFromScript	ScriptFile	Load a connector from a script
LoadProperties	PathToPropertyFile	Load a set of properties for your connector from a file
SetProperty	PropertyName PropertyValue	Set the value of a property
GetProperties		List all properties currently set on the connector
Start		Start the connector
Stop		Stop the connector

Table 5.5. Command Execution

Option	Arguments	Description
Select	Sql	Run a SELECT statement. This option takes multi-line input terminated with “;”
Insert	Sql	Execute an INSERT statement. This option takes multi-line input terminated with a “;”.
Update	Sql	Execute an UPDATE statement. This option takes multi-line input terminated with “;”
Delete	Sql	Execute a DELETE statement. This option takes multi-line input terminated with a “;”.
SetBatchSize	BatchSize	Set the batch size used when retrieving results
SetSecurityContext	VDBName VDBVersion UserName	Set the properties of the current security context
SetPrintStackOnError	PrintStackOnError	Set whether to print the stack trace when an error is received

Table 5.6. Scripting

Option	Arguments	Description
SetScriptFile	PathToScriptFile	Set the script file to use

Option	Arguments	Description
Run	ScriptName	Run a script with the file name
Runall		Run all scripts loaded by loadFromScript
RunScript	PathToScriptFile ScriptNameWithinFile	Run a particular script in a script file
SetFailOnError	FailOnError	Set whether to fail a script when an error is encountered or continue on
Result	ExpectedResults	Compares actual results from the previous command with the expected results. This command is only available when using the command line tester in script mode.

Table 5.7. Miscellaneous

Option	Arguments	Description
CreateArchive	PathTOArchiveFileName PathToCDKFileName PathToDirectoryForExtensionModules	Creates a connector archive file based on the properties supplied.
CreateTemplate	PathToTemplateFile	Create a template connector type file at the given file name.
Help		List all options
Quit		Quit the command line tester

5.4.2. Loading Your Connector

Preparing your connector to execute commands consists of the following steps:

1. Add your connector code to the CDK classpath. The cdk.bat script looks for this code in the CONNECTORPATH environment variable. This variable can be set with the DOS shell command “SET CONNECTORPATH=c:\path\to\connector.jar”. Alternately, you can modify the value of the CONNECTORPATH environment variable in the cdk.bat file.

2. Start the command line tester. You can start the tester by executing the cdk.bat file in the cdk directory of the Teiid Tools installation.

3. Load your connector class and the associated runtime metadata. You can load your connector by using the “load” command and specifying the fully-qualified class name of your Connector implementation and the path to a VDB file. The VDB runtime metadata archive should contain the metadata you want to use while testing.

4. Set any properties required by your connector. This can be accomplished with the setProperty command for individual properties or the loadProperties command to load a set of properties

from either a properties file or a connector binding file. You can use the “getProperties” command to view the current property settings.

5. Start the connector. Use the “start” command in the command-line tester to start your connector.

Following is an example transcript of how this process might look in a DOS command window. User input is in bold.

```
D:\metamatrix\console\cdk> set CONNECTORPATH=D:\myconn\myconn.jar
D:\metamatrix\console\cdk> cdk
========================= ENV SETTINGS =========================
MM_ROOT        = D:\metamatrix\console
MM_JAVA        = D:\metamatrix\console\jre
CONNECTORPATH  = D:\myconn\myconn.jar
CLASSPATH      = ;D:\metamatrix\console\cdk\metamatrix-cdk.jar;D:\myconn\myconn.jar;
================================================================

D:\metamatrix\console>D:\metamatrix\tools400wl7\console\jre\bin\java -Xmx256m -
Dmetamatrix.config.none -Dmetamatrix.log=4 com.metamatrix.cdk.ConnectorShell
Starting
Started
>load com.metamatrix.myconn.MyConnector d:\myconn\myconn.vdb
>setproperty user joe
>start
>
```

5.4.3. Executing Commands

Commands can be executed against your connector using the SELECT, INSERT, UPDATE, and DELETE commands. Procedure execution is not currently supported via the command line tester. Commands may span multiple lines and should be terminated with a “;”.

When a command is executed, the results are printed to the console. Following is an example session executing a SELECT command with the command line tester. User input is in bold.

```
>SELECT Name, Value FROM MyModel.MyGroup WHERE Name = ‘xyz’;
String Integer
xyz 5
xyz 10
>
```

5.4.4. Scripting

One of the most useful capabilities of the command-line tester is the ability to capture a sequence of commands in a script and automate the execution of the script. This allows for the rapid creation of regression and acceptance tests.

A script file may contain multiple scripts, where each script is grouped together with { } and a name. Following is an example of a script file. This script file also uses the special script-only command RESULTS that will compare the results of the last execution with the specified expected results.

```
test {
  load com.metamatrix.myconn.MyConnector d:\myconn\myconn.vdb
  setproperty user joe
  start

  SELECT Name, Value FROM MyModel.MyGroup WHERE Name = 'xyz';
  results [
  String Integer
  xyz 5
  xyz 10
  ]
}
```

To execute this file, run the command line tester in scripting mode and specify the script file and the script within the file:

```
D:\metamatrix\console\cdk>cdk runscript d:\myconn\my.script test
========================= ENV SETTINGS =========================
MM_ROOT        = D:\metamatrix\console
MM_JAVA        = D:\metamatrix\console\jre
CONNECTORPATH  = D:\myconn\myconn.jar
CLASSPATH      = ;D:\metamatrix\console\cdk\metamatrix-cdk.jar;D:\myconn\myconn.jar;
================================================================
```

```
D:\metamatrix\console>D:\metamatrix\tools400wl7\console\jre\bin\java -Xmx256m -
Dmetamatrix.config.none -Dmetamatrix.log=4 com.metamatrix.cdk.ConnectorShell runscript
 my.script
Starting
Started
>Executing: load com.metamatrix.myconn.MyConnector d:\myconn\myconn.vdb
>Executing: setproperty user joe
>Executing: start
>Executing: select Name, Value from MyModel.MyGroup where Name = 'xyz';
String Integer
xyz 5
xyz 15

>Test /metamatrix/tools400wl7/console/cdk/yahoo.script.test failed. CompareResults Error:
 Value mismatch at row 2 and column 2: expected = 10, actual = 15

>Finished
D:\metamatrix\console\cdk>
```

The script run above illustrates the output when the test result fails due to differences between expected and actual results. In this case the value was expected to be 10 in the script but was actually 15. The setFailOnError command can be used to fail the execution of the entire script if an error occurs.

Scripts can also be run in interactive mode by using the setScriptFile and run commands. This can be useful to record portions of your interactive testing to avoid re-typing later.

Connector Deployment

6.1. Overview

Once you have written and compiled the code for your connector, there are several steps to deploy your connector to a Teiid Server:

- Creating a Connector Type Definition file that defines the properties required to initialize your connector.
- Identifying the Extension Modules (jars and resources) required for the Connector to run.
- Creating the Connector Archive file to bundle the Connector Type Definition file and the Extension Modules.
- Importing the Connector Archive file in the Teiid Console.
- Creating a Connector Binding using your Connector Type.

This chapter will help you perform these steps.

6.2. Connector Type Definition File

A Connector Type Definition file defines a connector in the Teiid Server. The Connector Type Definition file defines some key properties that allow the Teiid Server to use your connector as well as specifying other properties your connector might need.

A Connector Type Definition file is in XML format and typically has the extension ".cdk". It defines a default name for the connector type, the properties expected by the connector, and other information that allows the properties to be displayed correctly in the Console when a Connector Binding is created from the Connector Type.

An example of this file can be found in *Appendix A.* It may be helpful to refer to this file while reading this section. The template file can also be created using the Connector Development Kit.

6.2.1. Required Properties

The Connector API requires the following properties for the Teiid Server to load and use your connector.

Table 6.1. Required Connector Properties

Property Attribute	Example Value	Description
ConnectorClass	com.my.connector.MyConnector	Fully qualified name of class implementing the Connector interface.
ConnectorClassPath	extensionjar:mycode.jar;	Semi-colon delimited list of jars defining the classpath of this connector. Typically this

Property Attribute	Example Value	Description
		includes the actual code for your connector as well as any 3rd party dependencies.

For more information on the Connector Classpath, see the section *Understanding the Connector Classpath*

6.2.2. Connector Properties

Most connectors will require some initialization parameters to connect to the underlying enterprise information system. These properties can be defined in the Connector Type Definition file along with their default values and other property metadata. The actual property values can be changed when the connector is deployed in the Teiid Console.

Each connector property carries with it several attributes that are used by the Teiid Console to integrate the connector seamlessly into the Teiid Server.

Table 6.2. All Properties

Property Name	Example Value	Description
Name	ExampleProperty	Property name – should only contain letters, no spaces or other punctuation. This is the name of the property as it will be passed to the connector in the ConnectorEnvironment.
DisplayName	Example property	The property name as displayed in the Console. Typically this is a nicely formatted version of the Name attribute.
ShortDescription	The example property is used to control something.	A short description that is displayed as a tooltip of the property in the Teiid Console.
DefaultValue	Xyz	A default value for the property. This value will be auto-filled when a connector binding is created from the Connector Type.
IsRequired	false	If true, then this property is required. Any required property without a value is displayed in red in the connector binding properties panel.
IsModifiable	true	If set to “false”, the property is visible only when viewing all properties and is not modifiable in the properties panel.
IsMasked	false	If set to “true”, the property will be masked with *’s when it is entered and saved in an encrypted form. This attribute is typically used with passwords.

Property Name	Example Value	Description
IsExpert	true	Depending on the property display, the property can be optionally displayed for advanced users.
PropertyType	String	The short name of a built-in Java primitive wrapper Object type. Other possible values include Integer, Boolean, etc.

A property may also be constrained to a set of allowed values by adding child AllowedValue elements, i.e. <AllowedValue>value</AllowedValue>. Adding allowed values will cause the property to be displayed with a dropdown that limits the user selection to the allowed values.

6.3. Extension Modules

6.3.1. Extension Modules

Extension Modules are used in the Teiid Server to store code that extends the Teiid Server in a central managed location. Extension Module JAR files are stored in the repository database and all Teiid processes access this database to obtain extension code. Custom connector code is typically deployed as extension models.

6.3.2. Understanding the Connector Classpath

Each connector is started in an isolated classloader instance. This classloader loads classes via the Teiid Extension Modules before loading classes from the Teiid system classpath. Ideally, all of your connector classes should be loaded from extension modules, which are configured in the Teiid Console.

The ConnectorClasspath property of your connector defines the extension module jars that are included in your connector's classpath. The connector classpath is defined as a semi-colon delimited list of extension modules. Extension module jar files must be prefixed with "extensionjar:"

6.4. Connector Archive File

The Connector Archive file is a bundled version of all files needed by this Connector to execute in the Teiid server. This file includes the Connector Type Definition file and all the Extension Modules required by the Connector to create a connector archive file (CAF)..

- The archive is a standard zip file.
- Start the CDK tool by executing cdk.bat
- Execute "CreateArchive" command by supplying:
  1. Path to the name of the archive file to create

2. Path to the Connector Type Definition file
3. Path to the directory where the required Extension Modules (jar files) are stored (note that only .jar files specified in the ConnectorClassPath property of the Connector Type definition file are bundled.

The file created by the CDK can be opened with any zip file utility to verify the required files are included.

The archive file can be tested in the CDK tool by loading it using the command “loadArchive”. Refer to Chapter 4 for more information on the CDK tool

6.5. Importing the Connector Archive

6.5.1. Into Teiid Server

To use a new connector type definition in the Teiid Server, the Connector Archive file must be imported in the Teiid Console or using the Admin API. To perform this task, perform the following steps:

1. Start the Teiid Console and connect to your Teiid Server.
2. Select Connector Types from the tree at the left of the Console. This will display a list of existing Connector Types on the right.
3. Click the Import… button on the bottom of the Connector Type list. This will open the Import Connector Type Wizard.
4. Select your Connector Archive file and click the Next button.
5. Click Finish to create the Connector Type. At this point you will see the new Connector Type in the list of Connector Types.
6. Select Extension Modules from the tree at the left of the Console, and make sure all the required Extension Modules are added.

6.5.2. Into Enterprise or Dimension Designer

To use the new connector type during the development of the VDB for testing using the SQLExplorer, Connector Archive File must be imported into the Designer tools. To perform this task, perform the following steps.

1. Start the Enterprise or Dimension designer
2. Open the project and in the “vdb” execute panel, click on the “Open the Configuration Manager” link. For more information consult the designer’s guide.

1. In the result window, click “Import a Connector Type (.cdk,.caf)” link and follow directions.

The Connector Type can now be used to create Connector Bindings.

6.6. Creating a Connector Binding

6.6.1. In Console

To actually use your connector in the Teiid System, you must create a Connector Binding that specifies the specific property values for an instance of the Connector Type. To create a Connector Binding, perform the following steps:

1. Start the Teiid Console and connect to your Teiid Server.
2. Select Connector Bindings from the tree at the left of the Console. This will display a list of existing Connector Bindings on the right.
3. Click the New… button below the list of Connector Bindings. This will launch the Create New Connector Binding Wizard.
4. In Step 1, you must specify a name for your connector binding and select your connector type from the Connector Type list. Click the Next button to continue.
5. In Step 2, the connector properties from your connector type definition file will be displayed. Default values are used to pre-fill the value fields if they exist. Required properties are displayed with bold text. Required properties with no value specified are displayed in red text. These fields must be completed before the Next button will enable. Optional properties may be displayed by checking the Optional Properties checkbox. When you have completed all required values, click the Next button.
6. In Step 3, you are given the opportunity to set the enabled state of the new binding in each PSC. Typically, no modifications need to be made. For more information, see the Teiid Console User's Guide. Click the Finish button to complete the wizard and create your connector binding. The Connector Binding list now displays your new connector binding.

To actually start your connector binding, please consult the Teiid Console User's Guide for detailed information.

6.6.2. In Designer

Connector Binding properties can also be defined in the Designer for the given Connector Type, if the corresponding Connector Archive File is imported into the Designer. If you try to execute your VDB with SQLExplorer in the Designer, this tool will present you with a window to specify such Connector Bindings. The user is required specify these binding properties before they can test using the SQLExplorer. For more information on how this can be accomplished please refer to the Enterprise Designer User's Guide.

Also, note that the bindings specified in the Designer tool are automatically bundled into the VDB for deployment, so if there are any properties that needs to be changed from development

environment to the production environment, those properties need to be modified when a VDB is deployed to the Teiid Server using the Console to correct resources.

Connection Pooling

7.1. Overview

The Query Engine logically obtains and releases a connection for each command that is executed.

However many enterprise sources maintain persistent connections that are expensive to create. For these situaions, Teiid provides a transparent connection pool to reuse rather than constantly release connections. The connection pool is highly configurable through configuration properties and extension APIs for Connections and Connectors

Many built-in connector types take advantage of pooling, including JDBC, Salesforce, and LDAP connectors.

7.2. Framework Overview

The table below lists the role of each class in the framework.

Table 7.1. Responsibilities of Connection Pool Classes

Class	Type	Description
PoolAwareConnection	Interface	This interface is an extension of the Connection interface and provides hooks to better interact with Connection pooling.
ConnectorIdentityFactory	Interface	Defines a factory for creating ConnectorIdentities. This can optionally be implemented by the concrete Connector class to properly segregate Connections in the pool. If this class is not implemented by the Connector, then SingleIdentity support will be assumed.
ConnectorIdentity	Interface	This interface corresponds to an identifier for a connection in the pool. Changing the identity implementation changes the basis on which connections are pooled. Connections that have equal identity objects (based on the equals() method) will be in the same pool.
SingleIdentity	Class	This implementation of the identity class makes all connections equivalent, thus creating a single pool.
UserIdentity	Class	This implementation of the identity class makes all connections equivalent for a particular user, thus creating a set of per-user connection pools.
ConnectionPooling	Annotation	This optional Annotation can be used on the Connector implementation class to indicate configure

Class	Type	Description
		pooling. This can be especially useful to indicate that automatic ConnectionPooing should not be used regardless of the connector binding property settings.

7.3. Using Connection Pooling

Automatic connection pooling does not require any changes to basic Connector development. It can be enabled by setting the Connector binding Property ConnectionPoolEnabled=true or by adding the ConnectionPooling annotation, which defaults to enabled=true, to the Connector implementation class. Automatic Connection pooling can be disabled if either setting is false.

Connector developers can optionally utilize the PoolAwareConnection and ConnectorIdentityFactory interfaces to refine the Connector's interactions with Connection pooling. It is important to consider providing an implementation for PoolAwareConnection.isAlive to indicate that a Connection is no longer viable and should be purged from the pool. Connection testing is performed upon leases from the pool and optionally at a regular interval that will purge idle Connections. It is also important to consider having the concrete Connector class implement ConnectorIdentity factory if Connections are made under more than just a single identity.

7.4. The Connection Lifecycle

These steps occur when connection pooling is enabled:

1. If the Connector implements ConnectorIdentityFactory, the ConnectorManager asks the Connector to generate a ConnectorIdentity for the given SecurityContext, else SingleIdentity is assumed. The ConnectorIdentity is then stored on the SecurityContext.
2. The ConnectorManager asks for a Connection from the pool that pertains to the ConnectorIdentity.
3. The ConnectionPool returns a Connection that was either pulled from the pool (and passes the isAlive check) or was created by the Connector if necessary.
4. After the ConnectorManager has used the Connection to execute a command, it releases the Connection. This call is intercepted by the pool and the method PoolAwareConnection.releaseCalled is invoked on the Connection instead. If the Connection does not implement PoolAwareConnection, it is assumed no action is needed.
5. When the Connection fails an isAlive check or becomes too old with pool shrinking enabled, it is purged from the pool and Connection.release is called.

7.4.1. XAConnection Pooling

The usage of XAConnections (that provide XAResources) typically come with additional limitations about how those Connections can be used once they are enlisted in a transaction. When enabled, automatic connection pooling will perform these additional features with XAConnections:

- The pool will return the same XAConnection for all executions under a given transaction until that transaction completes. This implies that all executions to a given XAConnector under the same connection will happen serially.
- XAConnections enlisted in a transaction will return to the pool once a transaction completes.
- Two separate pools will be maintained. One for Connections that have not and will not be used in a transaction, and one for XAConnections that have an will be used in a transaction. Each pool will be configured based upon the same set of configuration properties - it is not possible to independently control pool sizes, etc.

7.5. Configuring the Connection Pool

The ConnectionPool has a number of properties that can be configured via the connector binding expert properties.

Table 7.2. Connection Pool Properties

Name	Key	Default Value	Description
Connection Pool Enabled	ConnectionPoolEnabled		Explicitly enables or disables connection pooling.
Data Source Test Connect Interval (seconds)	SourceConnectionTestInterval	600	How often (in seconds) to create test connections to the underlying source to see if it is available.
Pool Maximum Connections	com.metamatrix.data. ~pool.max_connections	20	Maximum number of connections total in the pool. This value should be greater than 0.
Pool Maximum Connections for Each ID	com.metamatrix.data. ~pool.max_connections_for_each_id	20	Maximum number of connections per ConnectorIdentity object. This value should be greater than 0.
Pool Connection Idle Time (seconds)	com.metamatrix.data. ~pool.live_and_unused_time	60	Maximum idle time (in seconds) before a connection is closed if shrinking is enabled.
Pool Connection Waiting Time (milliseconds)	com.metamatrix.data. ~pool.wait_for_source_time	120000	Maximum time to wait (in milliseconds) for a connection to become available.
	com.metamatrix.data. ~pool.cleaning_interval	60	

Name	Key	Default Value	Description
Pool cleaning Interval (seconds)			Interval (in seconds) between checking for idle connections if shrinking is enabled.
Enable Pool Shrinking	com.metamatrix.data.~pool.enable_shrinking	true	Indicate whether the pool is allowed to shrink.

Monitored Connectors

8.1. Overview

The Teiid Connector API contains an optional interface that allows connectors to be automatically monitored by the Teiid Enterprise Server or checked via the Teiid Admin API.

8.2. Monitored Connector Framework Overview

This UML diagram shows the classes involved in the monitored connector classes.

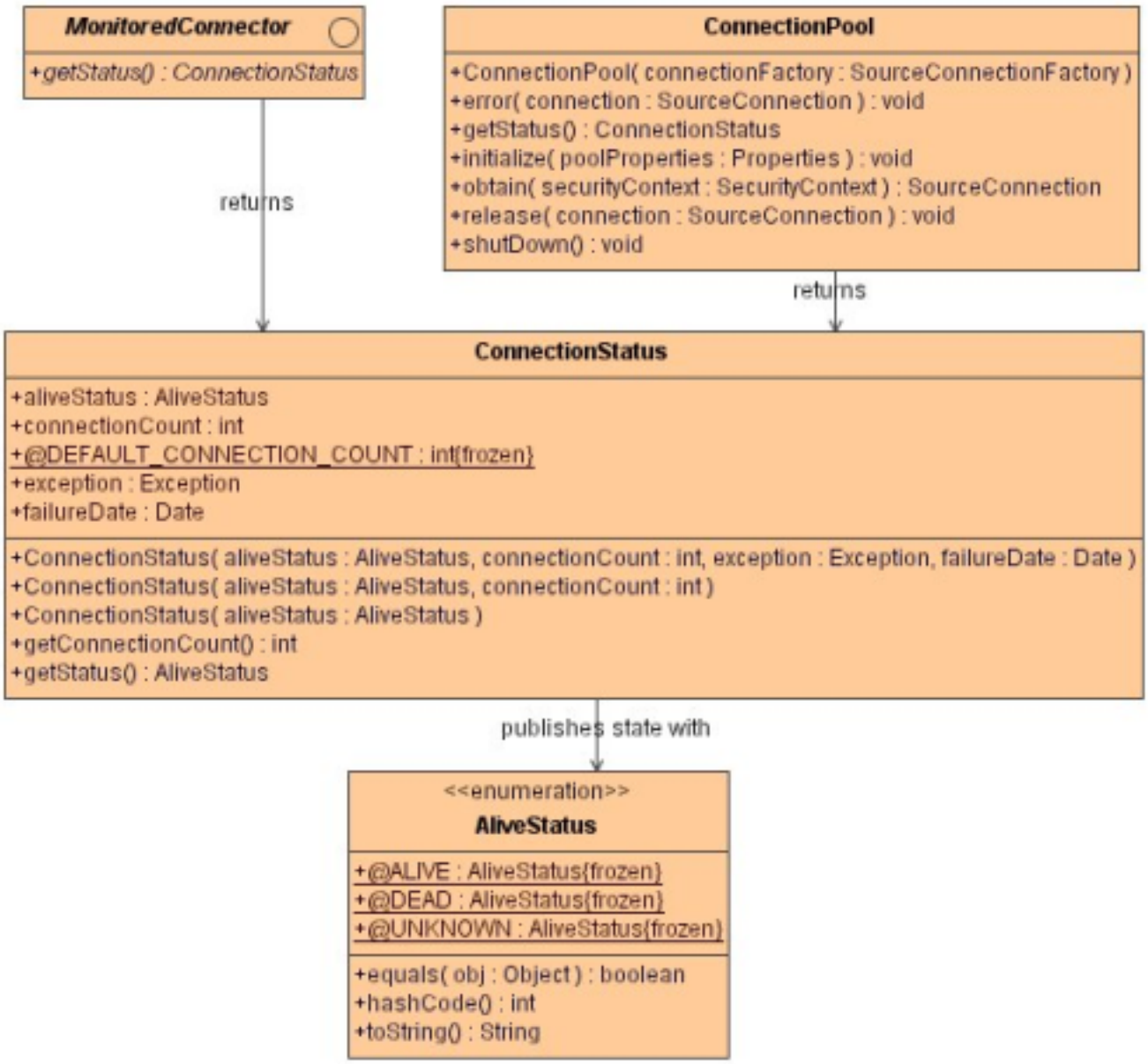

Figure 8.1. Monitored Connector Class Diagram

The table below lists the role of each class in the framework.

Table 8.1. Monitored Connector Classes

Class	Type	Description
MonitoredConnector	Interface	This interface can be added to the Connector implementation to indicate that the connector supports monitoring.
ConnectionStatus	Class	
AliveStatus	Class	This class defines an enumeration for valid status values for a ConnectionStatus.

8.3. Using The Framework

To support connector monitoring, your Connector implementation must extend the MonitoredConnector interface and implement the single getStatus() method to return a ConnectionStatus.

A monitored connector will be polled for status in the Teiid Enterprise Server (connector monitoring is not supported on Teiid Query or Dimension products). The poll rate is the value of the metamatrix.server.serviceMonitorInterval system property, which can be set in the Teiid Console. This property defaults to 60 seconds. If the ConnectionStatus indicates an AliveStatus of DEAD, then the connector is marked in the service registry as “data source unavailable”. If the ConnectionStatus indicates an AliveStatus of ALIVE, the connector is marked as “open”. An AliveStatus of UNKNOWN does not change the state of the registry.

In addition, the Admin API can be used in Teiid Query and Teiid Enterprise to obtain the status of connectors at runtime. For more information, see the Admin API Javadoc.

Handling Large Objects

This chapter examines how to use facilities provided by the Teiid Connector API to use large objects such as blobs, clobs, and xml in your connector.

9.1. Large Objects

9.1.1. Data Types

Teiid supports three large object runtime data types: blob, clob, and xml. A blob is a “binary large object”, a clob is a “character large object”, and “xml” is a “xml document”. Columns modeled as a blob, clob, or xml are treated similarly by the connector framework to support memory-safe streaming.

9.1.2. Why Use Large Object Support?

The Teiid Server allows a Connector to return a large object through the Teiid Connector API by just returning a reference to the actual large object. The Teiid Server or JDBC Driver can then access the data via a stream rather than retrieving the data all at once. This is useful for several reasons:

1. Reduces memory usage when returning the result set to the user.
2. Improves performance by passing less data in the result set.
3. Allows access to large objects when needed rather than assuming that users will always use the large object data.
4. Allows the passing of arbitrarily large data values within a fixed Teiid memory usage.

However, these benefits can only truly be gained if the Connector itself does not materialize an entire large object all at once. For example, the JDBC API supports a streaming interface for blob and clob data.

9.2. Handling Large Objects

The Connector API supports the handling of the large objects (Blob/Clob/SQLXML) through the creation of special purpose wrapper “type” objects. Each type of LOB object has a respective wrapper object.

Table 9.1. Lob Types

Java SQL Type	Runtime Type
java.sql.Blob	com.metamatrix.common.types.BlobType

Java SQL Type	Runtime Type
java.sql.Clob	com.metematrix.common.types.ClobType
com.metamatrix.core.sql.SQLXML	com.metematrix.common.types.XMLType

In the example below, the physical source returns an object type of Clob, then the connector should return the corresponding ClobType object.

```
//Example BatchedExecution.execute method

List columnValues = new ArrayList();

// building the reference
Clob clob = results.getClob();
ClobType clobReference = new ClobType(clob);
…
// this is needed to keep the connection open.
executionContext.keepExecutionAlive(true);

// adding the reference to batch of results
columnValues.add(clobReference);
batch.addRow(columnValues);
```

Once the wrapped object is returned, the streaming of LOB is automatically supported. These LOB objects then can be used to serve to client results, used in server for query processing, or used in user defined functions.

A connector execution is usually closed and the underlying connection is either closed/released as soon as all rows for that execution have been retrieved. However, LOB objects may need to be read after their initial retrieval of results. It is very important that the default closing behavior should be prevented to correctly stream the contents of the LOB based data. This behavior is communicated to the server through setting a flag in “ExecutionContext” interface by invoking

```
executionContext.keepExecutionAlive(true);
```

with this call, the server will close the connector execution object only after all returned LOB objects can no longer be read. i.e. when user Statement object is closed. Note that single call to keepExecutionAlive is needed per execution – and it must be called before the first batch is returned from connector

The SQLXML interface allows large xml documents to be processed by the server without creating memory issues. XML Source Connectors also use this interface to supply documents to the Teiid XQuery engine.

A new, and important, limitation of using the LOB type objects introduced in the 5.5 version of the Teiid Server is that streaming is not supported from remote connectors. This is an issue in clustered environments if connectors intended to return LOBs are deployed on only a subset of the hosts or in failover situations. The most appropriate workaround to this limitation is to deploy connectors intended to return LOBs on each host in the cluster. There is currently no workaround to support streaming LOBs from connectors in remote failover situations.

9.3. Inserting or Updating Large Objects

The Teiid JDBC API also allows the insertion or update of large objects. However, the JDBC API does not currently stream large objects on insert or update. So, the Teiid JDBC API will read all of the data and pass it back to the connector in a single materialized value.

In these cases LOBs will be passed to the Connector in the language objects as an ILiteral containing a java.sql.Blob, java.sql.Clob, or java.sql.SQLXML. You can use these interfaces to retrieve the data in the large object and use it for insert or update.

Appendix A. Connector Type Definition Template

This appendix contains an example of the Connector Type Definition file that can be used as a template when creating a new Connector Type Definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<ConfigurationDocument>
   <Header>
      <ApplicationCreatedBy>Connector Development Kit</ApplicationCreatedBy>
      <ApplicationVersionCreatedBy>4.0:1681</ApplicationVersionCreatedBy>
      <UserCreatedBy>MetaMatrixAdmin</UserCreatedBy>
      <DocumentTypeVersion>1.0</DocumentTypeVersion>
      <MetaMatrixSystemVersion>4.0</MetaMatrixSystemVersion>
      <Time>2008-01-30T15:22:05.296-06:00</Time>
   </Header>
   <ComponentTypes>
                        <ComponentType Name="My Connector" ComponentTypeCode="2"
 Deployable="true" Deprecated="false" Monitorable="false" SuperComponentType="Connector"
 ParentComponentType="Connectors">
     <!-- Required by Connector API -->
         <ComponentTypeDefn Deprecated="false">
                                                <PropertyDefinition Name="ConnectorClass"
  DisplayName="Connector Class" ShortDescription="" DefaultValue="com.mycode.Connector"
 IsRequired="true" IsMasked="false" IsModifiable="false" />
         </ComponentTypeDefn>
         <ComponentTypeDefn Deprecated="false">
                                                <PropertyDefinition Name="ConnectorClassPath"
    DisplayName="Class Path" ShortDescription="" DefaultValue="extensionjar:mycode.jar"
 IsRequired="true" IsMasked="false"  />
         </ComponentTypeDefn>

     <!-- Example properties - replace with custom properties -->
         <ComponentTypeDefn Deprecated="false">
                  <PropertyDefinition Name="ExampleOptional" DisplayName="Example Optional
 Property" ShortDescription="This property is optional due to not being marked as IsRequired"
 IsMasked="false" />
         </ComponentTypeDefn>
         <ComponentTypeDefn Deprecated="false">
```

```
        <PropertyDefinition Name="ExampleDefaultValue" DisplayName="Example Default Value
 Property" ShortDescription="This property has a default value" DefaultValue="Default value"
IsRequired="true"  IsMasked="false" />
        </ComponentTypeDefn>
        <ComponentTypeDefn Deprecated="false">
            <PropertyDefinition Name="ExampleEncrypted" DisplayName="Example Encrypted
 Property" ShortDescription="This property is encrypted in storage due to Masked=true"
IsRequired="true" IsMasked="true" />
        </ComponentTypeDefn>

        <ChangeHistory>
          <Property Name="LastChangedBy">ConfigurationStartup</Property>
          <Property Name="CreatedBy">ConfigurationStartup</Property>
        </ChangeHistory>
      </ComponentType>
  </ComponentTypes>
</ConfigurationDocument>
```