WebSphere software

IBM

WebSphere Process Server 6.1: Business Process Choreographer query() and queryAll()

How to access processes, tasks and work items through the API and JDBC

Rolf Bäurle, Frank Neumann
IBM Development Lab Boeblingen, Germany

December 2007

Table of Contents

Abstract

Business Process Choreographer is the process and human task engine in WebSphere® Process Server version 6.

Business processes are defined using WS-BPEL (Business Process Execution Language) and are deployed as part of a J2EE application in a WebSphere Process Server.

The Business Process Choreographer API offers access to entities (such as processes and tasks), stored in a runtime database.

This document explains technical details for the `query()` and `queryAll()` API functions and provides best practices on how to use them.

1 Introduction

During deployment of business process and human task applications in WebSphere Process Server, template information for entities, such as business processes, activities and tasks are created in the Business Choreographer runtime database (sometimes referred to as "BPEDB").

At runtime, Business Process Choreographer creates corresponding instance entities and manages these throughout their lifecycle.

A user may interact with an application based on Business Process Choreographer in order to retrieve a "todo" list of tasks that are ready to be claimed or to list process instance information. Therefore, the system must maintain a relationship between accessed objects (such as tasks) and the logged-in user.

This relationship between a person (or a group of persons), an entity and a particular reason is called a *work item*.

In addition to work list management, Business Process Choreographer leverages work items also for authorization and access control.

Understanding the concept of work items in Business Process Choreographer is therefore critical when interacting with the API, particularly with the `query()` and `queryAll()` API functions.

The Javadoc in the WebSphere Process Server Info Center [InfoCenter] and additional papers on the Business Process Choreographer programming model provide a good overview of the general concepts. The BPC Samples page [BPCSamples] has ready-to-use code samples that demonstrate interaction with Business Process Choreographer

This document provides conceptual and technical details for the `query()` and `queryAll()` implementation. Note however that the implementation of some details described in this document may change in future versions of the product without further notice.

2 Business Process Choreographer API

The Business Process Choreographer API comes in several renderings and is accessible from EJB-, Web Service- or JMS clients.

Examples in this document are based on the generic session EJB interface for the human task manager and the business flow manager, respectively.

In addition to API functions, a number of database views are published for direct access to entities and work items through a customer program, for example using JDBC or SQLJ.

Note that the human task manager API primarily offers access to task-related information and operations while the business flow manager API is designed to handle business process related requests. Because authorization and work items is a concept that spans both components, the respective APIs both include `query()` and `queryAll()` functions with comparable functionality.

The `query()` API call is designed to return information about entities in the context of an existing work item. You typically don't get entities that are not associated with work items. If multiple entities are referenced in a single `query()` API call, a defined order determines which entity must be correlated with a work item and thereby determines if a specific combination of entities is returned in the result set.

The `queryAll()` API call is designed to return information about all entities, regardless of an existing work item. This is particularly useful for monitoring or administrator purposes. Users calling this API function must be in a specific J2EE role (refer to 5.4).

Both the `query()` and `queryAll()` API operate in a SQL-like way on the underlying runtime database and return, quite similar to JDBC, a query result set with rows and columns (see section 6, The QueryResultSet).

The purpose of these API functions is to offer a programming model that is as close as possible to SQL and adds enhancements to shield the user from either database specific implementations or coherences internal to Business Process Choreographer.

A set of additional API functions are available that return entities and work items as Java™ objects. Please refer to the Javadoc in the Information Center [InfoCenter] for more details on related function such as `getAllWorkItems()` and `getAllActivities()`.

3 Work item management and authorization

Work items in Business Process Choreographer describe a relationship between an entity, a person (or a group or persons) and also include a particular reason.

The `WORK_ITEM` database view or the `com.ibm.task.api.WorkItem` interface gives a overview of the attributes (and also available values) of a work item.

They include

- A unique ID for the work item. As with other entities in the Business Process Choreographer API, an implementation of the `OID` interface is used to identify objects exposed through the API.
- The ID and type of the referenced entity. See the following section for details about which entities can have work items.
- The ID and type of an associated entity. This typically identifies an enclosing object (such as the business process) and is used, for example, for efficient cleanup purposed when a business process is deleted.
- Either one attribute of "owner id", "group name" and "everybody" is set to describe the relationship to the person(s) this work item is dedicated to.

Work items are used by Business Process Choreographer to allow access to entities, such as tasks and business processes.

If the principal name of a user who is logged onto WebSphere Process Server has at least one work item for an entity, the user is allowed to access its content.

A work item is considered to be assigned to a specific user if either

- The `owner_id` field of the work item contains the user's name
- The `everybody` attribute is set for a work item
- The `group_name` field contains the group name of a group the user belongs to (only valid if group work items are used)

Further checks in the Business Process Choreographer API and navigation engine for processes and tasks can restrict additional operations depending on the reason of a work item, however, to retrieve information using a `query()` operation, the existence of any work item is sufficient.

Note that the major difference between `query()` and `queryAll()` is the way how they handle authorization. While `query()` implements instance-based authentication with the help of work items, the `queryAll()` call requires membership in certain J2EE roles for authorization (refer to 5.4).

3.1 How and when work items get created

Work items are created for entities with a defined people assignment expression for human interaction. An example for such an assignment would be members of a certain LDAP group as the potential owners of a modeled human task.

In addition to modeled people assignment expressions, Business Process Choreographer also creates work items whenever human interaction is required. An example would be an error condition where an administrator needs to manually restart a process activity and therefore receives a work item.

Additional work items get created based on conceptual correlation between entities. For example:

- If an administrator of a business process should be able to manage its enclosed activities as well, additional work items for these activities are created.
- Inline human tasks (that is, tasks modeled in a business process) play both the role of tasks and activities. Creation of an additional activity work item allows the task administrator to manage the corresponding activity entity as well.

In general, Business Process Choreographer creates work items for the following entities:

- Human task (both inline and standalone)
- Human task templates
- Human task escalations
- Human task escalation templates
- Business processes
- Activities in a business process

- Events

Work items based on modeled people assignment expressions are created when the entity is created or enters the ready state.

Additional work items get created in the context of API interaction or subsequent processing of processes and task, where required.

People assignment expressions define, who (a person or a group of people) can perform a certain role for an entity. When an entity is activated, a work item with a corresponding reason is created for each qualifying user or group:

Reason	**Human Task**	**Escalation**	**Business Process**	**Activity**	**Event**
Potential Starter	X	-	X	-	-
Potential Owner	X	X	-	-	-
Instance Creator	X	-	X	-	-
Administrator*)	X	-	X	-	-
Editor	X	-	X	-	-
Reader	X	-	X	-	
Potential Sender	-	-	-	-	X
Escalation Receiver	-	X	-	-	-

In addition to these work items, the Business Process Choreographer navigation engine creates the following dedicated work items:

Reason	**Human Task**	**Escalation**	**Business Process**	**Activity**	**Event**
Starter	X	-	X	-	-
Originator	X	.	-	-	-
Administrator*)	X	-	-	X	-

*) An administrator work item for an inline human task is created for persons receiving an administrator work item for the enclosing business process. If an invoke activity goes into an error state, an administrator work item is created as well.

Since work items are used to implement authorization (the right to access and read the data for a certain entity), the additional administrator work items for an inline human task also allow the business process administrators to access the enclosed human task. See section 5.4**Error! Reference source not found.** for details.

3.2 Database tables for work items

During configuration of Business Process Choreographer, a relational database is associated with the business process container (sometime referred to as "BPEDB"). The database stores all template (model) and instance (run-time) data necessary to manage the business processes and tasks.

The Business Process Choreographer database schema contains tables that are used internally and published views (see section 3.3, Published queryable views).

Data relating to work items is stored in the following tables (note that table names in the Business Process Choreographer database schema usually end with `_T`, while views do not have this convention, also please note that names and content of internal tables are subject to change in future versions without further notice) :

- `WORK_ITEM_T`
  This table contains one row for a relationship between an entity (for example, a human task) and a person or a group of people. If more than one reason results in this relationship (for example, if a person is both administrator and starter of a process), one row is created for each reason.
- `RETRIEVED_USER_T`
  If the relationship is between an entity and a group of people (usually the result of a people assignment expression for a task), this table contains all qualifying users retrieved from the people directory provider for a certain people assignment expression.
- `WI_ASSOC_OID_T`
  Due to the hybrid way of dealing with inline human tasks as both tasks and process activities, this table is used to associate work items for human task to corresponding activities in a BPEL process. This table is also used to manage authorization for dependent entities.
- `STAFF_QUERY_TEMPLATE_T`
  During the deployment of tasks with people assignment expressions, the expressions are stored in a precompiled format in this table. If a process defines the same people assignment expression more than once, there is only one entry for this expression in the table, that is, if people assignment expressions are syntactically identical, they are shared in this table (if post-processing is disabled or sharing is explicitly enabled).
- `STAFF_QUERY_INSTANCE_T`
  If a people assignment expression is evaluated during task and business process lifecycle, an instance of the expression is created together with a timestamp when it expires. If the result of the people assignment evaluation is a single person or a group of people the user IDs are inserted into the `RETRIEVED_USER_T` table. Otherwise the everybody or group name information is stored in the `WORK_ITEM_`table. Subsequent instances of the same task or business process reuses the same staff query instance as long as it is independent from specific context or post-processing.

The WORK_ITEM view is defined as a join of the tables WORK_ITEM_T, WI_ASSOC_OID_T and RETRIEVED_USER_T to contain again one row per relationship and user, even if a group of people is assigned to a certain role. This view (together with additional published views) is the foundation for the query() API function.

3.3 Published queryable views

Since not only the work items themselves are of interest but also information about the referenced objects, such as human tasks and business processes, the query() API function also provides access to this information.

In order to do so (and to introduce an additional abstraction layer that allows the underlying tables to be changed in future versions of Business Process Choreographer), a couple of additional views are defined to be queryable by the query() and queryAll() API functions:

```
View name                  correlation name (see 5.2)
-------------------------------------------------------
WORK_ITEM                  WI
TASK                       TA
TASK_DESCR                 TAD
TASK_CPROP                 TACP
TASK_TEMPL                 TT
TASK_TEMPL_DESC            TTD
TASK_TEMPL_CPROP           TTCP
ESCALATION                 ESC
ESCALATION_DESC            ESCD
ESCALAION_CPROP            ESCCP
ESCALATION_TEMPL           ESCT
ESC_TEMPL_DESC             ETD
ESC_TEMPL_CPROP            ETCP
APPLICATION_COMPONENT      AC
PROCESS_TEMPLATE           PT
PROCESS_TEMPL_ATTR         PTA
PROCESS_INSTANCE           PI
PROCESS_ATTRIBUTE          PA
ACTIVITY                   AI
ACTIVITY_ATTRIBUTE         AA
ACTIVITY_SERVICE           AV
EVENT                      EI
QUERY_PROPERTY             QP
```

For a detailed list of the available columns/attributes for those views, see [InfoCenter].

The query() and queryAll() API calls care about the way and order how views are combined by adding appropriate join predicates. In order to return typed objects in the result set, further information about database column types are required by the underlying implementation.

If an application based on Business Process Choreographer wishes to reference custom tables or views in query() or queryAll() API calls, these custom tables need to be declared and registered with WebSphere Process Server. See chapter 11, Including custom tables or views, for more details.

3.4 Group work items

The previously described concept of assigning work items to persons results in a single entry in the work item view per qualifying user. This is a reasonable and efficient approach if only few users qualify for defined roles in a task people assignment expression and if membership of users to groups does not change frequently. Even in cases where a fair amount of users qualify, but the set of qualifying users is shared between multiple process instances, this is a reasonable approach because the amount of data created is limited.

If, on the other hand, thousands of users typically qualify for a task, assignment of users to groups change frequently or context variables in people assignment expressions prevent the resulting user list from being shared, the built-in concept of group work items is indicated.

With this option enabled in the human task container property page of the WebSphere admin console, the `group_name` column for work items is included and considered for work item queries.

Note that during process and task modeling in WebSphere Integration Developer, the corresponding people expression criteria must be used.

When deciding for group work items, a trade-off must be made between the number of users qualifying for a task and the number of groups the user belongs to. Due to a SQL `IN` predicate for the `group_name` column that is generated when processing work item queries, performance is impacted when a user belongs to many groups. In general, however, the group work item feature still outperforms regular people assignments when many users qualify for an entity's role.

3.5 Authority for business process admin and reader

While it may be obvious that a people assignment expression for a task results in corresponding work items for this object and for the particular reason, there are also cases where the creation of a single work item triggers creation of additional ones.

Users in the business process administrator and business process reader role also have "inherited" rights on the enclosed activities. Business Process Choreographer handles this in to ways: in some cases, additional work items are created for dependent objects (such as activities in a business process), in other cases additional predicates for the authorization term (see 5.4, Adding the authorization term – the system administrator role) are created to ensure that queries asking for both business process and activity data return results even if the logged-in user does not have a corresponding work item for the activity itself.

3.6 Authorization for the queryAll() API call

The `queryAll()` API call can only be invoked by users that are in the J2EE role `BPESystemAdministrator, BPESystemMonitor, TaskSystemAdministrator` or `TaskSystemMonitor`. Mapping of users or groups to these roles is defined during

installation and configuration of Business Process Choreographer and is a system wide setting. The WebSphere administrative console may be used to change the mapping for the business process and human task container.

4 Query clauses

4.1 Select clause

As in SQL, the select clause specifies which information is to be retrieved from the database system. The view and column names that are available are listed in [InfoCenter].

The select clause in Process Choreographer's `query()` API call must conform to the following syntax (for a complete description please refer to the JavaDoc [InfoCenter]) :

- The *select* clause can contain one or more column specifications, each of which describes one column of a queryable view.
- Multiple column specifications are separated by a comma ",".
- Each column specification must contain exactly one token of the form `view.column`, where `view` is one of the queryable views listed previously, and `column` is a column in this view. Each expression in the select clause containing a period "." is assumed to be a view-column token.
- The `view.column` token can be surrounded by any SQL "decoration" that is understood by the database system. This decoration, however, must not change the returned type. Aggregation functions `(MIN, MAX)`, for example, change the returned type and are therefore not supported. The `COUNT()` function however is supported and returns the number of qualified records as a numerical value.

If the table or column name is not recognized, a `QueryUnknownTableException` or `QueryUnknownColumnException` is thrown.

Examples of valid *select* clauses are:

- Get all work item IDs
  `"WORK_ITEM.WIID"`
- Get distinct task IDs and corresponding work item reason
  `"DISTINCT TASK.TKIID, WORK_ITEM.REASON"`
- Get process name and creation time
  `"PROCESS_INSTANCE.NAME, PROCESS_INSTANCE.CREATED"`
- Basic arithmetic, as long as it does not change the resulting column type and the database system supports the operation
  `"TASK.PRIORITY * 10"`
- Column aliases
  `"WORK_ITEM.REASON AS ROLE"`

Samples of **invalid** *select* clauses are:

- Aggregation functions (except `count`)
  `"MIN(ACTIVITY.CREATED)"`

- Type conversion (casting) that results in incompatible types
  `"CAST (ACTIVITY.CREATED AS CHAR)"`
- Unknown table or column names in a `view.column` token
  `"WORK_ITEM.DOES_NOT_EXIST"`
  `"MYVIEW.VALUE"`
- Expressions without a valid `view.column`, for example, returning a constant value, calling a stored procedure or user defined function
  `"WORK_ITEM.WIID, 'text', myfunction(xy)"`

4.2 Where clause

The *where* clause restricts the result set of the query and specifies filter criteria.

It is an optional parameter. If this parameter is null no filter is applied.

The *where* clause is processed in a similar way to the *select* clause. You must follow certain rules and most of the syntax of the underlying database system can be used for the expressions (for a complete description please refer to the JavaDoc [InfoCenter]):

- Each `view.column` token must refer to a known queryable view and column - as described in section 4.1, Select clause. If the table or column name is not recognized, a `QueryUnknownTableException` or `QueryUnknownColumnException` is thrown.
- The operands can be combined using the standard logical operators `AND` and `OR`.
- Subselects, which perform another full query in the *where* clause, are not generally supported. However, see section 4.10, Complex queries, for ways to get around this limitation.

Examples of valid *where* clauses are:

- Restrict query to all tasks in state `READY`
  `"TASK.STATE = TASK.STATE.STATE_READY"`
  where the syntax for the value on the right hand side is explained in section 4.9.
- Restrict query to all tasks in state `READY` and to work items with reason `POTENTIAL_OWNER` – the latter information is redundant because tasks in state `READY` always have a work item with reason `POTENTIAL_OWNER` but it might improve the query performance due to an additional filter
  `"TASK.STATE = TASK.STATE.STATE_READY AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER"`

Note: Certain column types, such as object IDs and timestamps, require a special syntax that is described later in this paper.

4.3 Order-by clause

The *order-by* clause can specify one or more columns that are used for ordering the result set. This is an optional parameter, and ordering is not applied if the parameter is null.

Processing the *order* clause is similar to what happens with the *select* clause, and the same syntactical restrictions apply.

You can specify the order direction by adding `descending` or `ascending` tokens.

The ordering operation is completely processed by the underlying database system, which means that the character order for certain locales (collation) and code pages must be specified there. Business Process Choreographer performs no post-processing when it creates the query result set.

4.4 Applying a threshold

Optionally, the number of rows fetched and returned in the `QueryResultSet` (refer to chapter 6) can be limited.

You should consider using this when the expected result is large and it is acceptable to limit the number of rows returned.

The implementation differs, and depends on what the underlying database system supports – see below.

Here are some examples that show how Business Process Choreographer uses this value to restrict the result set:

- For DB2 Universal Database® (DB2 UDB®), the resulting `SELECT` statement is restricted by a
  `"FETCH xxx FIRST ROWS ONLY"`
- For Oracle®, the *where* clause of the `SELECT` statement is extended by
  `"AND ROWNUM <= xxx"`
  In case a *order-by* clause is specified as well the resulting SQL statement for Oracle will be of the following form:
  `"SELECT * FROM (SELECT … ORDER BY YYY) WHERE ROWNUM <= xxx"`
  This way the result set will be ordered first and then the threshold will be applied.
- For all other database systems, the result set limitation is specified on the JDBC result set, which is, for example, the recommended way for Apache Derby.

4.5 Skipping tuples

The `skipTuples` parameter specifies the number of records from the beginning of a result set that are skipped and are not returned in the `QueryResultSet` object.

If the JDBC driver of the database system supports the `absolute()` method for the `java.sq.ResultSet` class the cursor on the result set can be moved to the given row number. Otherwise the cursor is moved using the `next()` method without materializing the row.
This parameter can be used together with the `threshold` parameter to implement paging in a client application, for example, to retrieve the first 20 items, then the next 20 items, and so on.

This is an optional parameter and no rows are skipped if this parameter is set to null.

4.6 Timestamps and time zones

The representation of times and dates normally depends on the current locale and time zone settings.

To support a multi-time zone environment, where a client application, the WebSphere Process Server, and the database back end can each reside in a different time zone, Business Process Choreographer has to take the different time zones into account, and provides ways to perform any necessary conversions.

Time information in the Business Process Choreographer database is stored in UTC (Universal Time Coordinated) in order to allow correlation of data written by multiple WebSphere Process Servers (potentially running in different time zones). A conversion might be required both before presenting time information on a user interface, and after getting time information from a user interface.

This conversion affects `query()`, `queryAll()` and `getTimestamp()` on the query result set.

The `query()` and `queryAll()` API calls expect timestamps in *where* clauses to be passed using a `TS('')` pseudo function, optionally a time zone can be specified. If no time zone is given (the parameter is null), timestamps are assumed to be in UTC and are stored in the database without any conversion.

Sample for passing a timestamp in UTC using the TS() function (`bfm` denotes the remote object for the business flow manager EJB API):

```
bfm.queryAll( "ACTIVITY.AIID",
           "ACTIVITY.STARTED > TS('2007-10-01T10:01:07')",
           (String) null, (Integer) null, (Integer) null, (TimeZone) null );
```

The same sample if the timestamp is in Pacific Standard Timezone (PST):

```
bfm.queryAll( "ACTIVITY.AIID",
           "ACTIVITY.STARTED > TS('2007-10-01T10:01:07')", (String) null,
           (Integer) null, (Integer) null, TimeZone.getTimeZone("PST"));
```

That is all you need to do in terms of time zone conversion to pass timestamp information from a client environment to Business Process Choreographer. The API functions convert the timestamp to UTC and continue working with the converted value.

The timestamp in the `TS('')` function is expected to be in the format:

```
YYYY-MM-DDThh:mm.ss
```

Everything except the year is optional, that is, can be omitted and is then set with default values. Thus, a timestamp expression only specifying the date part `TS('2007-10-01')` is valid, too.

Keep in mind, that if your multi-tier application runs on a process server processing requests from a remote web client, you must get the time zone information from the user's Web browser when passing it to the Query API.

The `getTimestamp()` function of the `QueryResultSet` returns a `java.util.Calendar` object that contains both time and time zone information. The time is always in UTC, independent of the time zone information. The time zone, that is set in the Calendar object is either UTC or the one passed in the previous `query()` or `queryAll()` call.

If you have a multi-tier environment where a Web client may have a different time zone setting compared to the machines where your JSP runs, this time zone information can help with the following conversion. If your code is running on the client machine, you can also ignore the time zone setting in the `Calendar` object and use the local settings.

To display time stamp information, you have to implement the proper time zone conversion and use a formatter class, such as `java.text.DateFormat` or `java.text.SimpleDateFormat`. The following sample shows how the value in a `Calendar` object can be displayed taking into account the time zone information that comes with the Calendar object (note that `htm` denotes the remote object of the human task manager EJB API):

```
resultSet = htm.query( "TASK.COMPLETED ", null, null, null, null,
                          TimeZone.getTimeZone("Europe/Berlin") );
  while( resultSet.next() )
  {
     Calendar cal = resultSet.getTimestamp( 1 );
     // target timezone as specified in query() (or UTC if null)
     // in this example, “Europe/Berlin”
     TimeZone tz = cal.getTimeZone();

     // Use either DateFormat or SimpleDateFormat
     DateFormat fmt = DateFormat.getDateTimeInstance(
                                 DateFormat.LONG, DateFormat.LONG );
     fmt.setTimeZone( tz );
     // Print the resulting
     System.out.println( "Task completed: " + fmt.format( cal.getTime() ) );
  }
```

The way that Business Process Choreographer fills the Calendar object when using a `QueryResultSet` object is the same for API objects like `ActivityInstanceData`, returned by an API call like `getActivityInstance(AIID)`. Since you do not pass a time zone parameter to this call, the Calendar object has the current local time zone set as a default.

4.7 Current date

Instead of explicitly passing the current date into a `TS()` function call of a *where* clause, the `CURRENT_DATE` token can be used to express this. This is important for defining work lists that should return a result that depends on the current time.

Most database systems also support arithmetic with timestamp columns. Because the `CURRENT_DATE` token is replaced with a parameter marker (please refer to section 5.1) in further processing, adding and subtracting time to the `CURRENT_DATE` token is not supported, however, you can specify additional time information with the comparing column.

The following example shows how to use `CURRENT_DATE` in *where* clauses with DB2 Universal Database syntax (the syntax for other database systems might differ):

```
"... TASK.COMPLETED + 3 DAYS > CURRENT_DATE ..."
```

Example of invalid use:

```
"... TASK.COMPLETED > CURRENT_DATE – 3 DAYS"
```

Note that `CURRENT_DATE` is a token that is interpreted by Business Process Choreographer and replaced with the current timestamp (in UTC) on the WebSphere Process Server. Similar expressions such as `CURRENT_TIMESTAMP` are provided by database systems like DB2 UDB and are resolved with the current (local) system time on the database server.

4.8 Object IDs

To identify application objects, such as activities or business processes, identifiers (IDs) are introduced for most of these objects. In the database, IDs are represented as 16 bytes of raw (binary) data.

To handle IDs in application programs, there is also a string representation that can be used in a *where* clause. A process instance, for example, can be identified by an ID, such as:

```
_PI:800300f3.9aee33.9e1ced53.530c00a0
```

To pass this ID in a qualifying *where* clause of a `query()` or `queryAll()` call, it has to be specified using the ID() pseudo function:

```
… AND PROCESS_INSTANCE.PIID = ID('_PI:800300f3.9aee33.9e1ced53.530c00a0') .
```

The ID() pseudo function translates the ID string representation back to its binary format and passes it to the underlying database system. Again, the corresponding SQL statement uses parameter markers and replaces correlation names:

```
… AND PI.PIID = ?
```

If an ID is selected in the result set, either `getOID()` or `getString()` can be applied to the result set. The first returns an `OID` object, the second returns the corresponding string representation. If you have the choice of working either with `OID` objects or with strings, you should use the `OID` representation, because passing an object is considerably faster than transforming a string representation.

Note: If you omit the `ID()` pseudo function for object IDs, and the database system performs an implicit type conversion from an object ID string to the binary database representation, you do not get an SQL error. In this case, you will only notice that the result set is empty and no rows are returned.

It is important to check your *where* clauses carefully to ensure object IDs are handled correctly.

4.9 Syntax for constant values

Some columns in queryable views are defined as numeric, but should only contain certain, distinct values. This is comparable to an enumeration in programming languages.

Instead of comparing these column values with the concrete integer value, it is good practice to compare these values against predefined constants.

For example, the `WORK_ITEM.REASON` column has one of the following values:

- `REASON_POTENTIAL_OWNER=1`
- `REASON_EDITOR=2`
- `REASON_READER=3`

- REASON_OWNER=4
- REASON_POTENTIAL_STARTER=5
- REASON_STARTER=6
- REASON_ADMINISTRATOR=7
- REASON_POTENTIAL_SENDER=8
- REASON_ORIGINATOR=9
- REASON_ESCALATION_RECEIVER=10
- REASON_POTENTIAL_INSTANCE_CREATOR=11

Assume you are interested in all work items for objects for which you have an owner work item (the ones you claimed).

A valid *where* clause for this query would be:

```
" ... WORK_ITEM.REASON = 4 ..."
```

Since this obscures the meaning, it is recommended that you use the predefined constant values with the following syntax:

```
" ...WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER ..."
```

Note: The constant expression on the right is constructed according to the format:

```
<view>.<column>.<constant name>
```

The constant name must be fully qualified in the sense that it is has to be scoped by `view.column` to avoid problems with constants defined for other columns.

Another way to replace fixed numerical values by the more flexible named constants is to use the values defined in the corresponding API object:

```
" ... WORK_ITEM.REASON = " + WorkItemData.REASON_OWNER + "..."
```

The corresponding API objects does not only contain all defined constant expressions as integer values, but also have additional valuable information for all columns.

You may check out the entity class (for example, `WorkItemData`) in the Java documentation or explore the defined symbolic names in WebSphere Integration Developer by inspecting the corresponding Java class.

4.10 Complex queries

As already mentioned, sub queries are, in general, not supported. However, if you know the rules and algorithms used, you can run sub queries.

Consider the common requirement for querying the currently valid business process template.

Each process template includes a `VALID_FROM` field that specifies the timestamp when the template becomes valid. To discover which template is selected by Business Process Choreographer if a new process with name processTempl1 is started, see the following example. It shows a *where* clause passed to a `queryProcessTemplate()` API call (see 9, queryProcessTemplates() and queryTaskTemplates()), which is very similar to the `query()` API call except that it returns `ProcessTemplate` API objects rather than a `QueryResultSet`:

```
"PROCESS_TEMPLATE.NAME = 'processTempl1' AND PROCESS_TEMPLATE.VALID_FROM =
(SELECT MAX(VALID_FROM) FROM PROCESS_TEMPLATE WHERE
NAME=PROCESS_TEMPLATE.NAME AND VALID_FROM <= CURRENT_DATE )"
```

The subselect clause returns the maximum value of VALID_FROM, not later than the current date. Understanding how the name comparison, NAME=PROCESS_TEMPLATE.NAME works, requires knowledge from the previous sections of this document: This clause gets transformed using the PT alias for the PROCESS_TEMPLATE view and parameter markers:

```
"PT.NAME = ? AND PT.VALID_FROM = (SELECT MAX(VALID_FROM) FROM
PROCESS_TEMPLATE WHERE NAME=PT.NAME AND VALID_FROM <= ?)"
```

Note that the VALID_FROM column in MAX() and NAME column at the end are not replaced because they are not recognized and do not contain a period ".". The same is true for the table name PROCESS_TEMPLATE in the *from* clause.
Also note that the MAX() function can be used here because it is placed inside the subselect and therefore it does not change the returned type (refer to 4.1).

Because PROCESS_TEMPLATE.NAME is replaced by PT.NAME, the query returns the desired result.

5 Query processing

The parameters of the query() and queryAll() API call in Business Process Choreographer are designed to be as similar as possible to SQL syntax. Deviations and enhancements are mainly to assist in describing Business Process Choreographer specifics (such as ID handling).

The implementation in Business Process Choreographer does no full SQL parsing but rather applies an efficient scanning of the passed parameters.

The resulting SQL statement can be introspected by enable traces for Business Process Choreographer (the generated JDBC statements are actually pretty close to what is specified in the parameters for query() and queryAll()).

In general, processing of query calls includes the following steps:

- Collecting referenced views and replacing view names with defined correlation names (see 5.2)
- Replacing literals (strings and numbers) and constant values with parameter markers
- Parsing timestamps and IDs, and converting them to JDBC values in a prepared statement
- Adding an appropriate *from* clause to the SQL statement
- Adding join information between the referenced views (either inner join or left outer join)
- Optionally adding a result set limitation threshold value (depending on the database system)

The result is a JDBC prepared statement that is processed by the database system.

The details of these steps are discussed later in this document.

If the optional `skipTuples` parameter is used, appropriate steps are applied to skip rows in the JDBC result set. Depending on the capabilities of the JDBC driver, skipping rows is supported natively or must be emulated by just reading but not materializing rows of the JDBC result set.

It is important to be aware that Business Process Choreographer does not parse the clauses passed to it because parsing is performed by the database system. This allows the SQL syntax that is available for the database system to be fully exploited, rather than requiring a restricted subset supported by all possible database systems.

5.1 Parameter markers and prepared statements

Each literal expression, such as a string or a numerical constant value, that is found in the *where* clause of a `query()` API call is replaced by a parameter marker.

For example, the following comparison expression in a *where* clause:

```
"... TASK.ORIGINATOR = 'Frank' ..."
```

is replaced using a parameter marker:

```
"... TASK.ORIGINATOR = ? ..."
```

Applications, such as a Web client, normally run a limited set of queries, differing only in string literals (for example, consider a query to retrieve all work items belonging to the logged in user).

Without the use of prepared statements, these queries would have to be compiled in the database system for each user, causing a significant performance impact.

After replacing parameter markers, however, these queries are syntactically identical, and the database system only needs to build an access plan once, and can reuse it for subsequent queries.

5.2 Correlation names

To shorten the SQL statement and to allow for complex queries (see 4.10), correlation names are assigned to all queryable views (refer to list in chapter 3.3).

A query, such as

```
queryAll("TASK.NAME", "TASK.ORIGINATOR = 'Frank'", (String) null,
         (Integer) null, (Integer) null, (TimeZone) null);
```

will result in the following SQL statement:

```
SELECT TA.NAME FROM TASK TA WHERE TA.ORIGINATOR = ?
```

where the parameter marker will be set with the given value 'Frank' when the SQL statement is executed.

5.3 From clause, join algorithm, join hierarchy

For those familiar with SQL, it might seem strange that *select* clauses and *where* clauses must be given to the `query()` API call, while the *from* clause is omitted.

The reason is that the *from* clause, which specifies the views to be queried is computed based on the referenced views in the *select*, *where*, and *order-by* clauses. While these clauses are scanned, the set of referenced views is built that becomes the basis for the *from* clause.

For the `query()` API call, the `WORK_ITEM` view plays a special role in this process because it is always added to the set of views, even if none of the query() parameters references it explicitly.

In addition to computing the *from* clause of the resulting SQL statement, the necessary join predicates for these views are added to the *where* clause. Business Process Choreographer uses internally defined dependencies of views and the matching join columns to compile the join predicate. The *where* clause of a `query()` API call must typically not contain join predicates – they either collide with the ones that are automatically generated or are (in the best case) just superfluous.

For example, consider the following *select* clause (assuming both *where* and *orderBy* clauses are not set) where a user wants to query the state of all tasks for which she/he has a work item, using the `query()` API call:

```
selectClause = "TASK.STATE"
```

Internally, the SQL *from* clause is formed by all referenced views, which in this example is only `TASK`, plus the `WORK_ITEM` view:

```
fromClause = "WORK_ITEM WI, TASK TA"
```

The *where* clause gets the proper join predicate appended – in this case, leveraging the knowledge that the work item object id for referenced task entities contains the task's `TKIID`:

```
whereClause = "WI.OBJECT_ID = TA.TKIID"
```

While this appears to be a rather straightforward operation, it can become more complex if multiple entities are involved.

Consider the following *select* clause (again for simplicity, the *where* clause is assumed to be not set) where the user wants to do the same query as above but is also interested in the name of the enclosing process instance for inline human tasks:

```
selectClause = "TASK.STATE, PROCESS_INSTANCE.NAME"
```

The SQL *from* clause becomes:

```
fromClause = "WORK_ITEM WI, TASK TA, PROCESS_INSTANCE PI"
```

However, what about the additional join predicates for the *where* clause?

Business Process Choreographer uses a built-in hierarchy of entities (the so-called *join level)* and knowledge about which columns to use for the join predicate that fits the user's expectation:

```
whereClause = "WI.OBJECT_ID = TA.TKIID AND
          TA. CONTAINMENT_CTXT_ID = PI.PIID"
```

An alternative (rejected) join predicate could have been:

```
"WI.OBJECT_ID=PI.PIID AND PI.PIID=TA. CONTAINMENT_CTXT_ID"
```

However, this would have returned information about all tasks where the user has a work item for the enclosing process rather than having a work item for the task, which is

undesirable and does not fit the authorization concept in Business Process Choreographer.

5.3.1 Outer join for optional data

By default, Business Process Choreographer adds join predicates for referenced database views using an inner join. This is desirable if the result should contain attributes from multiple views and the result only makes sense if a corresponding entry exists for all referenced views. As an example, a query referencing both (inline) human tasks attributes and business process attributes is expected to return a result if an inline task and its corresponding business process can be found.

However, there are view combinations where additional information (such as a task's description in a specific language) is optional.

If the desired user interface contains both attributes from the task view and the task's description for a given language, the result should return the task information, even if no corresponding description exists. Hence, the expected behavior is different and depends on the views that are joined.

Technically speaking, for join predicates with optional data, a SQL outer join is generated. Business Process Choreographer defines a couple of views that are considered to be optional join partners (mainly process, activity and task attributes, query properties and descriptions for tasks and escalations) and automatically generates outer joins for these in the right way.

5.3.2 Multiple references to a view

Now think about how to generate a query that returns information about processes for that two given custom attributes are set, for example *Street* and *City*. Using the following *where* clause

```
where ="PROCESS_ATTRIBUTE.NAME='Street' AND PROCESS_ATTRIBUTE.NAME='City';
```

a SQL statement is generated that – when executed - will not return the expected result because there is no attribute that is named *Street* **and** *City* as well (an attribute can only have a single name).

To solve this problem the custom attribute view has to be referenced twice for this example, once for the attribute *Street* and once for the attribute *City*.

In order to reference such multiple occurrences of a view in the *select*, *where* and *order* clauses, an index number must be added at the end of the view name.

For example, the following *select* and *where* clause can be used to query process instance information along with values of custom property *Street* (referenced with index 1) and *City* (referenced with index 2).

```
selectClause ="DISTINCT PROCESS_INSTANCE.PIID,
               PROCESS_ATTRIBUTE1.VALUE, PROCESS_ATTRIBUTE2.VALUE";
whereClause  ="PROCESS_ATTRIBUTE1.NAME='Street' AND
               PROCESS_ATTRIBUTE2.NAME='City';
```

5.4 Adding the authorization term – the system administrator role

As described in section 3, Work item management and authorization, work items are used for both authorization and assignment of "todos".

Technically, this authorization concept is realized by adding corresponding SQL predicates to the *where* clause that make sure that:

`WORK_ITEM.OWNER_ID` equals to the principal name of the logged in user

or

`WORK_ITEM.EVERYBODY` is set to true

or

`WORK_ITEM.GROUP_NAME` is included in the security context of the logged in user[1].

With the previously described mechanism of adding the `WORK_ITEM` view and proper join predicates, this completes the instance-based authorization concept for the `query()` API call in Business Process Choreographer.

This concept implies that regular users solely get associated data for which they have work items. If the user does not have a work item assigned for the object, no corresponding data can be retrieved using the `query()` API call.

There is however an exception to this rule: for users in J2EE role `BPESystemAdministrator` or `TaskSystemAdministrator`, the previously mentioned authorization term is *not* added. As a result, a work item (actually, any work item) must still exist, but it may be assigned to any user, not necessarily the one in J2EE role `BPESystemAdministrator` or `TaskSystemAdministrator`. The concept is, that users in these special roles administer the system and therefore need to access all entities in the system, even those for which they are not explicitly authorized.

It is important not to confuse the roles assigned during modeling (such as "administrator" or "potential owner") with the J2EE roles assigned during deployment. Process/task model roles are defined on per process/task basis while the J2EE role assignments are defined for the Business Process Choreographer J2EE applications and also grant enhanced access to additional API functions.

Because Process Choreographer uses the principal name of the user logged on to WebSphere Process Server for database comparisons, you must be careful if the Application Server uses a case-insensitive directory for authentication, for example, the user registry with an Application Server running on Windows. Although the user can log on to the server with different case spellings of her logon name, the `query()` call might return no results because the underlying database system performs a case-sensitive comparison for the principal name. In such an environment, you must either handle this

[1] This part is only added if the *group work item* feature is enabled.

in the client code, or make the users aware of the need to match cases exactly when entering their logon user ID.

5.5 Query execution isolation level

For internal process or task navigation Business Process Choreographer uses the JDBC transaction isolation level `TRANSACTION_REPEATABLE_READ` (for DB2 UDB that matches to 'Read Stability') to execute nearly all of its SQL statements to guarantee data consistency.

On the other hand a common work item query might read a lot of records in the database tables and therefore this transaction isolation level would establish many database locks and thus increase the risk for lock waits or deadlocks.

To minimize the impact for the process and task navigation Business Process Choreographer uses the transaction isolation level `TRANSACTION_READ_UNCOMMITTED` for work item queries – this approach guarantees best query performance and a minimum of database locks.

6 The QueryResultSet

Both the `query()` and `queryAll()` functions of the human task manager and business flow manager APIs return a query result set comprising rows and columns of the requested result.

Example of a `query()` API call and result set processing:

```
import com.ibm.task.api.*;
   ...
   InitialContext initialContext = new InitialContext();

   // lookup the EJB home interface
   Object object =
      initialContext.lookup("com/ibm/task/api/HumanTaskManager");
   HumanTaskManagerHome htmHome = (HumanTaskManagerHome)
      javax.rmi.PortableRemoteObject.narrow(object,
      HumanTaskManagerHome.class);

   // get the remote interface
   HumanTaskManager htm = htmHome.create();

   // query all tasks that are ready to be claimed
   // and the enclosing process instance's name
   QueryResultSet resultSet = htm.query(
      // select clause - what do we want to get?
      "PROCESS_INSTANCE.NAME, TASK.NAME",
      // where clause - what are the qualifying rows?
      "WORK_ITEM.REASON=WORK_ITEM.REASON.REASON_POTENTIAL_OWNER AND " +
     "TASK.STATE = TASK.STATE.STATE_READY",
      // order clause - specify the sort order
      "PROCESS_INSTANCE.NAME",
      // threshold - return first 10 entries only
      new Integer(10),
```

```
    // no timezone specified
    (TimeZone)null
);

// loop over results in the result set
while( resultSet.next() )
{
    // print out selected columns,
    // keep in mind column indexes start with "1"
    System.out.println( "Process instance name = " +
                        resultSet.getString(1) );
    System.out.println( "Task name = " + resultSet.getString(2) );
}
```

Note, that a `QueryResultSet` is similar to a JDBC `ResultSet` object, particularly with regard to cursor-driven navigation and column indexes starting with "1" instead of "0":

- After a `query()` or `queryAll()` call, the result set cursor is positioned before the first entry.
- Applying `next()` moves the cursor one line down and returns `false` if the end of the result set is reached.
- `first()` and `last()` can be used to revisit already read entries.
- `size()` returns the number of rows in the result set

However, in contrast to a JDBC `ResultSet`, a `QueryResultSet` object is serializable and can be sent to a remote client. It also includes Business Process Choreographer specific enhancements, such as ID and timestamp handling.

Navigating a `QueryResultSet` is always based on an in-memory copy of the result; therefore the returned result set does not change when moving forward and backward, even if the underlying data in the database is modified concurrently.

6.1 Reading QueryResultSet data

The `QueryResultSet` knows seven column types, defined in class `QueryColumnInfo`:

- `TYPE_STRING`
  For any text string columns, this type is also used for columns stored in CLOBs.
- `TYPE_NUMBER`
  For any numerical data, such as short, integer, or big integer.
- `TYPE_TIMESTAMP`
  For any timestamp data. The resolution is normally milliseconds (where supported by the database system).
- `TYPE_BINARY`
  For binary strings and BLOB data.
- `TYPE_BOOLEAN`
  For storing boolean values (`true` or `false`), the database representation is a `smallint` column.

- `TYPE_ID`
  For entity identifiers – each entity is identified by one or more of these identifiers, for example, process instance ID (`PIID`) and activity instance ID (`AIID`). They are stored in the database as a 16 byte long binary field.
- `TYPE_DECIMAL`
  For any numerical data that can be represented as a floating point number, such as double, integer etc.

The `QueryResultSet` provides functions to read data from the current cursor position

- `byte [] getBinary()`
  Returns a column of `TYPE_BINARY`. For example, a `VARBINARY` column.
- `Boolean getBoolean()`
  Returns "true", "false", or null. This function can also be applied to any numeric column type and returns true for any non-null value.
- `Integer getInteger(), Long getLong(), Short getShort(), Double getDouble()`
  Returns the corresponding numerical value or null.
  Note: These functions can be applied to all numerical types, however, if the underlying data type is different, its type is "casted" which might result in a loss in precision.
- `Object getObject()`
  Returns a generic object for a column value. You can use this function if you do not know the type or if it is only needed to be passed to another function.
- `OID getOID()`
  If the column is an identifier, this function returns an `OID` object that is needed for other API calls. You may have to cast the `OID` (base interface) to the more specific `OID` class, such as `PIID` or `AIID`.
- `String getString()`
  Returns a string representation of the specified column. This function is not only applicable to character based database columns, it can also be used to get a string representation of numerical values, timestamps, and `OID` values. For constant values, this function returns the descriptive name of the constant rather than the numerical value.
- `Calendar getTimestamp()`
  Returns the timestamp value for the specified column. This function takes into account the current client time zone setting. For details, see Timestamps and time zones.

If a column index is specified that is outside the range of the number of columns, an `IndexOutOfBoundsException` (runtime) exception is thrown.

If the function is not applicable to the corresponding column type, a `ClassCastException` is thrown.

The safest way to avoid these exceptions is to examine the column types first and then call the appropriate data accessor function on the query result set.

6.2 QueryResultSet meta data

Normally, a program calling the query() API function knows which column in the database it wants to query. However, if a user is allowed to specify arbitrary columns, a generic GUI application must find out what the result set contains before it displays the data.

For this purpose, the `QueryResultSet` class offers the following functions to get meta data about the selected columns:

- `numberColumns()`
  Returns the number of columns in the select clause.
- `getColumnDisplayName(int columnIndex)`
  Returns the name of the column, which is useful for displaying headings on a result table.
- `getTableDisplayName(int columnIndex)`
  Returns the name of the table or view. This is useful for displaying headings on a result table or to clarify to which view a column belongs.
- `getColumnType(int columnIndex)`
  Returns the type of the column. This function returns one of the constants defined for column types in the `QueryColumnInfo` interface: `TYPE_STRING`, `TYPE_NUMBER`, `TYPE_TIMESTAMP`, `TYPE_BINARY`, `TYPE_BOOLEAN`, `TYPE_ID`, `TYPE_DECIMAL`.
- `size()`
  Returns the number of entries in the result set. This can be useful for allocating memory before reading the result set data.

The following sample demonstrates how to display result set contents where the structure (columns) are unknown because, for example, the query clauses have been passed dynamically from user input. Note that an enhanced approach could also investigate the column type and use the typed accessors of the `QueryResultSet` object instead.

```
void displayResultSet( QueryResultSet resultSet )
   {
       // Print table heading
       for( int i=0; i<resultSet.numberColumns(); i++ )
       {
          System.out.print( resultSet.getColumnDisplayName(i) );
          System.out.print( "\t" );
       }
       System.out.println();

       // Print row data (String representations)
       while( resultSet.next() )
       {
          for( int i=0; i<resultSet.numberColumns(); i++ )
          {
             System.out.print( resultSet.getString(i) );
             System.out.print( "\t" );
          }
          System.out.println();
```

```
        }
    }
```

7 Stored queries

Stored queries are pre-defined queries that are stored in the Business Process Choreographer database. When they are processed they behave like any other ordinary query that is run by the `query()` API.
There are two kinds of stored queries:

- public stored queries are visible and available to all users.
  If different users run the same public stored query they usually will get different result sets depending on the objects they are authorized to see - that is, they have work items for.
- private stored queries are owned by a specific user and can only be executed by this owner and the system administrator.

A stored query is identified by its name and ownership. This way a public and a private stored query can have the same name. Also private stored queries that are owned by different users can have the same name.

Stored queries can be augmented with parameters to increase reusability. These parameters are resolved at runtime when the stored query is processed.

For example, the following *where* clause will query tasks that are named 'MyTask':

```
"TASK.NAME = 'MyTask'
```

To make this query reusable so that you can also search for other task names you can define the query using parameters:

```
"TASK.NAME = '@param1'
```

The parameter will be resolved at runtime by the parameter value that is passed to the query method.

8 People assignment expiration and caching

For performance reasons, Business Process Choreographer caches the user IDs retrieved from the registered and configured people directory provider during the evaluation of people queries. In addition to this, the results of syntactically equivalent people queries are shared within one process model (for further details please refer to [InfoCenter]).

By default, the list of cached user IDs is assumed to be valid for one hour. After this duration a result is considered to be expired. It is also possible to override the default expiration time for cached people query results of one hour.

The cached people query results are kept as long as the corresponding task or process template exists in the database. Once a business process application is uninstalled and thus the containing task and/or process templates are removed from the database the people query results are deleted as well.

If context variables are used in a people assignment criteria that prevent the results from being shared or work items are frequently transferred the cached result list might become that huge that query performance is negatively impacted. For those scenarios the `cleanupUnsedStaffQueryInstances.py` script should be called regularly to remove all people query results from the database that are currently not referenced by any work item.

The same caching applies if the same people assignment expression was defined for multiple inline human tasks within a business process, for example, where there are two tasks, and the same group of people is allowed to claim the task.

There are several ways to refresh the people query results in the database.

8.1 Refresh people query results using the administrative console

On the configuration tab of the Human Task container you can refresh all cached people query results in the database. Note that this might cause high load on your database system depending on the number of people assignments that are cached in the database.

8.2 Refresh people query results using administrative commands

Using the `refreshStaffQuery.py` administration script you can refresh cached people query results that satisfy some criteria.

For example, it is possible to refresh only those people query results that belong to a certain task template or that contain some specified user names.

It is also possible to refresh all cached results as described in 8.1

8.3 Refresh people query results using the refresh daemon

The refresh daemon is a process that checks all cached people query results regularly for currency. All cached people query results that are expired are refreshed.
The refresh daemon can be configured on the Human Task container. You can specify the schedule by a WebSphere CRON calendar entry.

9 queryProcessTemplates() and queryTaskTemplates()

The `query()` and `queryAll()` API calls described previously are designed to query work items and/or associated instance objects such as process instances and human task instances.

To get information about business process and task templates, Business Process Choreographer provides the `queryProcessTemplates()` and `queryTaskTemplates()` methods.

The reason for providing additional API calls for business process templates and task templates is because work items are typically not created for template data. Thus, evaluating the access rights for a process template (mainly the `REASON_POTENTIAL_STARTER`) is computed differently and do not require a join operation with the `WORK_ITEM` view.

Furthermore, the `queryProcessTemplates()` and `queryTaskTemplates()` API calls are simpler than `query()`, because they return the entire `ProcessTemplateData` respectively `TaskTemplateData` API objects and not just the columns and attributes specified in a *select* clause.

The *where* and *order-by* clauses, *threshold*, and *timezone* parameters are processed in the same way as in the `query()` API call, though.

10 Using views with JDBC

In addition to access Business Process Choreographer data through the API, the published database views can also be directly accessed with JDBC Java programs (or any other technology, such as SQLJ).

Advantages running JDBC queries include:

- More flexibility in the way how views can be aggregated and joined
- Support for native database system functions, such as UDFs and stored procedures
- May run in environments with no Business Process Choreographer access
- May run in J2SE environments
- Inclusion of and reference to database objects not belonging to Business Process Choreographer's known database schema

Downsides of using JDBC rather than query():

- No automatic joining of views
- No built-in security concept based on work items
- No database-system independent interface (SQL syntax slightly varies between database systems)

10.1 Retrieving a database connection

When running in the context of a WebSphere Process Server, a JNDI lookup is used to retrieve the datasource for Business Process Choreographer's runtime database ("BPEDB"). Within a J2EE environment, J2EE applications should define and use resource references in order to access a data source.

Authorization may be defined via authentication alias on the data source or by passing user credentials in the database connect request.

Following the resource reference of the Business Process Choreographer data source as defined in the deployment descriptor of the business process respective task container:

```
<resource-ref>
   <res-ref-name>jdbc/BPEDB</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Application</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

Referring to this entry a data source handle can be obtained via a JNDI lookup call, e.g.:

```
// JNDI lookup via comp/env
InitialContext ic = new InitialContext();
javax.sql.DataSource ds =
    (javax.sql.DataSource) ic.lookup("java:comp/env/jdbc/BPEDB");
// get a connection from the datasource
java.sql.Connection con = ds.getConnection();
```

It is important to use the J2EE programming model provided by the WebSphere Application server in order to benefit from quality of service, such as connection pooling and participation in global transaction.

If your application runs in a J2SE environment, either the data source class of the JDBC driver or an instance of `DriverManager` may be used to retrieve a connection to the database.

The above example illustrates how the existing Business Process Choreographer data source could be used for your own application. If you plan to change data source settings, it is highly recommended to define another dedicated data source for your client application to avoid any impact on workflow or task processing.

10.2 Running a JDBC select statement

Following a sample for a simple JDBC based query that retrieves all tasks that have been started by a given user ID:

```
// get a JDBC connection to the Business Process Choreographer
// runtime database ("BPEDB")
java.sql.Connection con = ds.getConnection();
// create SQL statement string
String sql = "SELECT TA.NAME FROM TASK TA WHERE TA.STARTER = ?";
// prepare a JDBC statement
java.sql.PreparedStatement stmt = con.prepareStatement(sql);
// set the parameter value
stmt.setString(1, "Rolf");
// execute the query
java.sql.ResultSet result = stmt.executeQuery();
// loop over result set and print task name to stdout
while ( result.next())
{7
  System.out.println( result.getString(1));
}
result.close();
stmt.close();
con.close();
```

Note that the above sample code does not include the logic to retrieve the data source handle (refer to 10.1) nor does it demonstrate appropriate exception and error condition handling.

11 Including custom tables or views

This section describes how own custom tables or views can be registered and used with Business Process Choreographer API functions.

This might be helpful for applications combining process or task data stored in the Business Process Choreographer runtime database with additional business data provided by other systems or application code.

In order to fit into the existing Business Process Choreographer programming model, both the logical structure and the correlation with published views must be defined and declared. A custom table definition file is used to describe the layout and characteristics of a custom table.

This custom table definition file is then registered in WebSphere Process Server.

The following sections guides through the steps of defining and registering a sample table.

A simple scenario could just query a custom table using the `queryAll()` API function. This table might be populated using triggers based on tables in the same database. The benefit over direct JDBC access to the table is a common programming model and database-independent helper functions, such as ID- and timestamp handling.

A more complex scenario could also include multiple custom tables that are combined with multiple Business Process Choreographer views.

11.1 An example

For this document, we assume that an additional table named `CUSTOM_DATA` is created in the Business Process Choreographer runtime database, for example using the following DDL statement in DB2 UDB syntax:

```
CREATE TABLE CUSTOM_DATA (
       ID             CHAR (16) FOR BIT DATA NOT NULL,
       NAME           VARCHAR (128),
       STREET         VARCHAR (128),
       ZIP            INTEGER,
       CITY           VARCHAR (128)
)
```

This `CUSTOM_DATA` table is populated by another application and we assume that the `ID` column contains the process instance `ID` (`PIID`) of a running business process in Business Process Choreographer.

A `queryAll()` API call is used to retrieve a process instance list that contains both Business Process Choreographer runtime data as well as data extracted from the `CUSTOM_DATA` table. The following code snippet assumes that a remote EJB interface to the business flow manager API (`bfm`) has been retrieved before.

```
String selectClause = "PROCESS_INSTANCE.NAME,
                       PROCESS_INSTANCE.STARTER,
                       CUSTOM_DATA.NAME, CUSTOM_DATA.CITY";
String whereClause =  "CUSTOM_DATA.ZIP = '71034'";

QueryResultSet result = bfm.queryAll( selectClause, whereClause, … );
```

During processing of the `queryAll()` API call, Business Process Choreographer adds a join predicate that includes the `PIID` column for `PROCESS_INSTANCE` and the `ID` column for `CUSTOM_DATA`. The result set therefore contains data for process instance and corresponding additional attributes from the `CUSTOM_DATA` table.

11.2 Defining a custom table

The structure of the table to register must be described in an XML file based on the schema definition in `bpecommon.jar/com/ibm/bpe/database/customtable.xsd`. For the example in the previous section, such a custom table definition file might be created as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<customtable xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/schemas/workflow/wswf/customtable
http://www.ibm.com/schemas/workflow/wswf/customtable"
xmlns="http://www.ibm.com/schemas/workflow/wswf/customtable">

  <querytableinfo tablename="CUSTOM_DATA" aliasname="CD" joinlevel="3">

    <joincolumn column="ID"/>
    <joincolumn column="ID" target="PROCESS_INSTANCE "/>

    <querycolumninfo
          columnname="ID"     type="TYPE_ID"     isNullable="false"/>
    <querycolumninfo
          columnname="NAME"   type="TYPE_STRING" isNullable="true"/>
    <querycolumninfo
          columnname="STREET" type="TYPE_STRING" isNullable="true"/>
    <querycolumninfo
          columnname="ZIP"    type="TYPE_NUMBER" isNullable="true"/>
    <querycolumninfo
          columnname="NUMBER" type="TYPE_STRING" isNullable="true"/>
  </querytableinfo>
</customtable>
```

The attributes in the `<querytableinfo>` element describe the custom table's name and an alias name that is used in the resulting SQL statement.

If multiple views are referenced in a `query()` or `queryAll()` API call, the `joinlevel` attribute helps to find the view that should be used for a join with the custom table.

The following table lists recommended values for specific scenarios:

Join with Business Process Choreographer view	Recommended `joinlevel` value
Process instance attribute	1
Process instance	3

Task instance	8
Work item	9

In addition to the `joinlevel` value, registration requires specification of the column used to generate a corresponding join predicate. The `<joincolumn>` element defines the column name that is generally used for join predicates. Additional `<joincolumn>` elements with a `target` attributes may be used to specify the column to be used for a specific target view.

In our example, we wish to combine process instance data with our custom data and therefore use a `joinlevel` attribute of `3` and specify `ID` as the columns that should be used to join with the `PROCESS_INSTANCE` view.

The `<querycolumninfo>` tags declare columns of the table or view that can be referenced in the `query()/queryAll()` API call. Note that not all columns of a table need to be registered. In our example, we choose to declare all columns (`ID, Name, Street, ZIP and City`) of the `CUSTOM_DATA` table.

The available `type` attribute values (such as `TYPE_ID, TYPE_STRING`) are defined in class `QueryColumnInfo` (see Javadoc in the WebSphere Process Server InfoCenter). Both `type` and `isNullable` attribute values and must match the underlying table or view definition.

The custom table definition file may contain multiple custom tables.

11.3 Registering a custom table

The custom table definition file described in the previous section must be stored in a file system location that is accessible by WebSphere Process Server.

In a clustered environment, this file (or a copy of it) must be stored in the same location for each server.

Make sure that your WebSphere Process Server is up and running. Open the administration console and navigate to `Servers → Application servers → server1 → Business process container.`

On the business process container panel select `Custom Properties.`

In the list of custom properties for the business container click "New" and add a new entry named `customTableDefinition` with a value that specifies the file system location of the previously created custom table definition file (for example, `d:\wps\customData.xml`).

Restart your WebSphere Process Server for the new setting to be picked up.

Note that this registration affects the `query()/queryAll()` API functions for both the business flow manager (BFM) API and the human task manager (HTM) API respectively, even though registration is solely through the business process container.

You are now ready to use your registered custom table together with the built-in Business Process Choreographer views.

12 Related references

[BPCSamples] Business Process Choreographer Samples
http://publib.boulder.ibm.com/bpcsamp/index.html

[InfoCenter] WebSphere Process Server Info Center
http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp

13 About the authors

The authors of this paper, Rolf Bäurle (baeurle@de.ibm.com) and Frank Neumann (frank_neumann@de.ibm.com) both work in the Business Process Choreographer development team at the IBM Germany Development Lab, Böblingen.

14 Trademarks