7

Oracle Database Advanced Queuing Operations Using PL/SQL

These topics describes the Oracle Database Advanced Queuing (AQ) PL/SQL operational interface.

> **See Also:**
>
> - Oracle Database Advanced Queuing: Programmatic Interfaces for a list of available functions in each programmatic interface
> - "DBMS_AQ" in *Oracle Database PL/SQL Packages and Types Reference* for more information on the PL/SQL interface
> - *Oracle Database Advanced Queuing Java API Reference* for more information on the Java interface
> - "More OCI Relational Functions" in *Oracle Call Interface Programmer's Guide*
> - "OCI Programming Advanced Topics" in *Oracle Call Interface Programmer's Guide* for more information on the Oracle Call Interface (OCI)

Using Secure Queues

For secure queues, you must specify the `sender_id` in the `messages_properties` parameter.

See "MESSAGE_PROPERTIES_T Type" in *Oracle Database PL/SQL Packages and Types Reference* for more information about `sender_id`.

When you use secure queues, the following are required:

- You must have created a valid Oracle Database Advanced Queuing agent using `DBMS_AQADM.CREATE_AQ_AGENT`.

- You must map `sender_id` to a database user with enqueue privileges on the secure queue. Use `DBMS_AQADM.ENABLE_DB_ACCESS` to do this.

> **See Also:**
>
> – "Creating an Oracle Database Advanced Queuing Agent"
> – "Enabling Database Access"

Enqueuing Messages

This procedure adds a message to the specified queue.

```
DBMS_AQ.ENQUEUE(
   queue_name          IN      VARCHAR2,
   enqueue_options     IN      enqueue_options_t,
   message_properties  IN      message_properties_t,
   payload             IN      "type_name",
   msgid               OUT     RAW);
```

It is not possible to update the message payload after a message has been enqueued. If you want to change the message payload, then you must dequeue the message and enqueue a new message.

To store a payload of type `RAW`, Oracle Database Advanced Queuing creates a queue table with `LOB` column as the payload repository. The maximum size of the payload is determined by which programmatic interface you use to access Oracle Database Advanced Queuing. For PL/SQL, Java and precompilers the limit is 32K; for the OCI the limit is 4G.

If a message is enqueued to a multiconsumer queue with no recipient and the queue has no subscribers (or rule-based subscribers that match this message), then Oracle error ORA 24033 is raised. This is a warning that the message will be discarded because there are no recipients or subscribers to whom it can be delivered.

If several messages are enqueued in the same second, then they all have the same `enq_time`. In this case the order in which messages are dequeued depends on `step_no`, a variable that is monotonically increasing for each message that has the same `enq_time`. There is no situation when both `enq_time` and `step_no` are the same for two messages enqueued in the same session.

Enqueue Options

The `enqueue_options` parameter specifies the options available for the enqueue operation. It has the following attributes:

- `visibility`

  The `visibility` attribute specifies the transactional behavior of the enqueue request. `ON_COMMIT` (the default) makes the enqueue is part of the current transaction. `IMMEDIATE` makes the enqueue operation an autonomous transaction which commits at the end of the operation.

  Do not use the `IMMEDIATE` option when you want to use LOB locators. LOB locators are valid only for the duration of the transaction. Your locator will not be valid, because the `immediate` option automatically commits the transaction.

You must set the `visibility` attribute to `IMMEDIATE` to use buffered messaging.

- `relative_msgid`

  The `relative_msgid` attribute specifies the message identifier of the message referenced in the sequence deviation operation. This parameter is ignored unless `sequence_deviation` is specified with the `BEFORE` attribute.

- `sequence_deviation`

  The `sequence_deviation` attribute specifies when the message should be dequeued, relative to other messages already in the queue. `BEFORE` puts the message ahead of the message specified by `relative_msgid`. `TOP` puts the message ahead of any other messages.

  Specifying `sequence_deviation` for a message introduces some restrictions for the delay and priority values that can be specified for this message. The delay of this message must be less than or equal to the delay of the message before which this message is to be enqueued. The priority of this message must be greater than or equal to the priority of the message before which this message is to be enqueued.

  > **Note:**
  >
  > The `sequence_deviation` attribute has no effect in releases prior to Oracle Database Advanced Queuing 10*g* Release 1 (10.1) if `message_grouping` is set to `TRANSACTIONAL`.
  >
  > The sequence deviation feature is deprecated in Oracle Database Advanced Queuing 10*g* Release 2 (10.2).

- `transformation`

  The `transformation` attribute specifies a transformation that will be applied before enqueuing the message. The return type of the transformation function must match the type of the queue.

- `delivery_mode`

  If the `delivery_mode` attribute is the default `PERSISTENT`, then the message is enqueued as a persistent message. If it is set to `BUFFERED`, then the message is enqueued as an buffered message. Null values are not allowed.

Message Properties

The `message_properties` parameter contains the information that Oracle Database Advanced Queuing uses to manage individual messages. It has the following attributes:

- `priority`

  The `priority` attribute specifies the priority of the message. It can be any number, including negative numbers. A smaller number indicates higher priority.

- `delay`

  The `delay` attribute specifies the number of seconds during which a message is in the `WAITING` state. After this number of seconds, the message is in the `READY` state and available for dequeuing. If you specify `NO_DELAY`, then the message is available for immediate dequeuing. Dequeuing by `msgid` overrides the `delay` specification.

> **Note:**
>
> Delay is not supported with buffered messaging.

- `expiration`

  The `expiration` attribute specifies the number of seconds during which the message is available for dequeuing, starting from when the message reaches the `READY` state. If the message is not dequeued before it expires, then it is moved to the exception queue in the `EXPIRED` state. If you specify `NEVER`, then the message does not expire.

> **Note:**
>
> Message delay and expiration are enforced by the queue monitor (`QMN`) background processes. You must start the QMN processes for the database if you intend to use the delay and expiration features of Oracle Database Advanced Queuing.

- `correlation`

  The `correlation` attribute is an identifier supplied by the producer of the message at enqueue time.

- `attempts`

  The `attemps` attribute specifies the number of attempts that have been made to dequeue the message. This parameter cannot be set at enqueue time.

- `recipient_list`

  The `recipient_list` parameter is valid only for queues that allow multiple consumers. The default recipients are the queue subscribers.

- `exception_queue`

  The `exception_queue` attribute specifies the name of the queue into which the message is moved if it cannot be processed successfully. If the exception queue specified does not exist at the time of the move, then the message is moved to the default exception queue associated with the queue table, and a warning is logged in the alert log.

- `delivery_mode`

  Any value for `delivery_mode` specified in message properties at enqueue time is ignored. The value specified in enqueue options is used to set the delivery mode of the message. If the delivery mode in enqueue options is left unspecified, then it defaults to persistent.

- `enqueue_time`

  The `enqueue_time` attribute specifies the time the message was enqueued. This value is always in Universal Coordinated Time (UTC), and is determined by the system and cannot be set by the user at enqueue time.

> **Note:**
>
> Because information about seasonal changes in the system clock (switching between standard time and daylight-saving time, for example) is stored with each queue table, seasonal changes are automatically reflected in `enqueue_time`. If the system clock is changed for some other reason, then you must restart the database for Oracle Database Advanced Queuing to pick up the changed time.

- `state`

  The `state` attribute specifies the state of the message at the time of the dequeue. This parameter cannot be set at enqueue time.

- `sender_id`

  The `sender_id` attribute is an identifier of type `aq$_agent` specified at enqueue time by the message producer.

- `original_msgid`

  The original_msgid attribute is used by Oracle Database AQ for propagating messages.

- `transaction_group`

  The `transaction_group` attribute specifies the transaction group for the message. This attribute is set only by `DBMS_AQ.DEQUEUE_ARRAY`. This attribute cannot be used to set the transaction group of a message through `DBMS_AQ.ENQUEUE` or `DBMS_AQ.ENQUEUE_ARRAY`.

- `user_property`

  The `user_property` attribute is optional. It is used to store additional information about the payload.

The examples in the following topics use the same users, message types, queue tables, and queues as do the examples in Oracle Database Advanced Queuing Administrative Interface. If you have not already created these structures in your test environment, then you must run the following examples:

- Example 12-1
- Example 12-2
- Example 12-3
- Example 12-5
- Example 12-7
- Example 12-8
- Example 12-23
- Example 12-25
- Example 12-26
- Example 12-27
- Example 12-28
- Example 12-36

For Example 12-1, you must connect as a user with administrative privileges. For the other examples in the preceding list, you can connect as user `test_adm`. After you have created the

queues, you must start them as shown in "Starting a Queue". Except as noted otherwise, you can connect as ordinary queue user 'test' to run all examples.

Enqueuing a LOB Type Message

Example 7-3 creates procedure blobenqueue() using the test.lob_type message payload object type created in Example 12-1. On enqueue, the LOB attribute is set to EMPTY_BLOB. After the enqueue completes, but before the transaction is committed, the LOB attribute is selected from the user_data column of the test.lob_qtab queue table. The LOB data is written to the queue using the LOB interfaces (which are available through both OCI and PL/SQL). The actual enqueue operation is shown in

On dequeue, the message payload will contain the LOB locator. You can use this LOB locator after the dequeue, but before the transaction is committed, to read the LOB data. This is shown in Example 7-14.

Enqueuing Multiple Messages to a Single-Consumer Queue

Example 7-5 enqueues six messages to test.obj_queue. These messages are dequeued in Example 7-17.

Enqueuing Multiple Messages to a Multiconsumer Queue

Example 7-6 requires that you connect as user 'test_adm' to add subscribers RED and GREEN to queue test.multiconsumer_queue. The subscribers are required for Example 7-7.

Example 7-7 enqueues multiple messages from sender 001. MESSAGE 1 is intended for all queue subscribers. MESSAGE 2 is intended for RED and BLUE. These messages are dequeued in Example 7-17.

Enqueuing Grouped Messages

Example 7-8 enqueues three groups of messages, with three messages in each group. These messages are dequeued in Example 7-16.

Enqueuing a Message with Delay and Expiration

In Example 7-9, an application wants a message to be dequeued no earlier than a week from now, but no later than three weeks from now. Because expiration is calculated from the earliest dequeue time, this requires setting the expiration time for two weeks.

Example 7-1 Enqueuing a Message, Specifying Queue Name and Payload

```
DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             test.message_typ;
BEGIN
   message := test.message_typ(001, 'TEST MESSAGE', 'First message to obj_queue');
   DBMS_AQ.ENQUEUE(
      queue_name              => 'test.obj_queue',
      enqueue_options         => enqueue_options,
      message_properties      => message_properties,
      payload                 => message,
      msgid                   => message_handle);
   COMMIT;
END;
/
```

Example 7-2 Enqueuing a Message, Specifying Priority

```
DECLARE
   enqueue_options       DBMS_AQ.enqueue_options_t;
   message_properties    DBMS_AQ.message_properties_t;
   message_handle        RAW(16);
   message               test.order_typ;
BEGIN
   message := test.order_typ(002, 'PRIORITY MESSAGE', 'priority 30');
   message_properties.priority := 30;
   DBMS_AQ.ENQUEUE(
      queue_name              => 'test.priority_queue',
      enqueue_options         => enqueue_options,
      message_properties      => message_properties,
      payload                 => message,
      msgid                   => message_handle);
   COMMIT;
END;
/
```

Example 7-3 Creating an Enqueue Procedure for LOB Type Messages

```
CREATE OR REPLACE PROCEDURE blobenqueue(msgno IN NUMBER) AS
   enq_userdata          test.lob_typ;
   enq_msgid             RAW(16);
   enqueue_options       DBMS_AQ.enqueue_options_t;
   message_properties    DBMS_AQ.message_properties_t;
   lob_loc               BLOB;
   buffer                RAW(4096);
BEGIN
   buffer       := HEXTORAW(RPAD('FF', 4096, 'FF'));
   enq_userdata := test.lob_typ(msgno, 'Large Lob data', EMPTY_BLOB(), msgno);
   DBMS_AQ.ENQUEUE(
      queue_name          => 'test.lob_queue',
      enqueue_options     => enqueue_options,
      message_properties  => message_properties,
      payload             => enq_userdata,
      msgid               => enq_msgid);
   SELECT t.user_data.data INTO lob_loc
      FROM lob_qtab t
      WHERE t.msgid = enq_msgid;
   DBMS_LOB.WRITE(lob_loc, 2000, 1, buffer );
   COMMIT;
END;
/
```

Example 7-4 Enqueuing a LOB Type Message

```
BEGIN
   FOR i IN 1..5 LOOP
      blobenqueue(i);
   END LOOP;
END;
/
```

Example 7-5 Enqueuing Multiple Messages

```
SET SERVEROUTPUT ON
DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
```

```
   message             test.message_typ;
BEGIN
   message := test.message_typ(001, 'ORANGE', 'ORANGE enqueued first.');
   DBMS_AQ.ENQUEUE(
         queue_name           => 'test.obj_queue',
         enqueue_options      => enqueue_options,
         message_properties   => message_properties,
         payload              => message,
         msgid                => message_handle);
   message := test.message_typ(001, 'ORANGE', 'ORANGE also enqueued second.');
   DBMS_AQ.ENQUEUE(
         queue_name           => 'test.obj_queue',
         enqueue_options      => enqueue_options,
         message_properties   => message_properties,
         payload              => message,
         msgid                => message_handle);
   message := test.message_typ(001, 'YELLOW', 'YELLOW enqueued third.');
   DBMS_AQ.ENQUEUE(
         queue_name           => 'test.obj_queue',
         enqueue_options      => enqueue_options,
         message_properties   => message_properties,
         payload              => message,
         msgid                => message_handle);
   message := test.message_typ(001, 'VIOLET', 'VIOLET enqueued fourth.');
   DBMS_AQ.ENQUEUE(
         queue_name           => 'test.obj_queue',
         enqueue_options      => enqueue_options,
         message_properties   => message_properties,
         payload              => message,
         msgid                => message_handle);
   message := test.message_typ(001, 'PURPLE', 'PURPLE enqueued fifth.');
   DBMS_AQ.ENQUEUE(
         queue_name           => 'test.obj_queue',
         enqueue_options      => enqueue_options,
         message_properties   => message_properties,
         payload              => message,
         msgid                => message_handle);
   message := test.message_typ(001, 'PINK', 'PINK enqueued sixth.');
   DBMS_AQ.ENQUEUE(
         queue_name           => 'test.obj_queue',
         enqueue_options      => enqueue_options,
         message_properties   => message_properties,
         payload              => message,
         msgid                => message_handle);
   COMMIT;
END;
/
```

Example 7-6 Adding Subscribers RED and GREEN

```
DECLARE
   subscriber         sys.aq$_agent;
BEGIN
   subscriber     :=  sys.aq$_agent('RED', NULL, NULL);
   DBMS_AQADM.ADD_SUBSCRIBER(
      queue_name  =>  'test.multiconsumer_queue',
      subscriber  =>  subscriber);

   subscriber     :=  sys.aq$_agent('GREEN', NULL, NULL);
   DBMS_AQADM.ADD_SUBSCRIBER(
      queue_name  =>  'test.multiconsumer_queue',
      subscriber  =>  subscriber);
```

```
END;
/
```

Example 7-7 Enqueuing Multiple Messages to a Multiconsumer Queue

```
DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   recipients          DBMS_AQ.aq$_recipient_list_t;
   message_handle      RAW(16);
   message             test.message_typ;
BEGIN
   message := test.message_typ(001, 'MESSAGE 1','For queue subscribers');
   DBMS_AQ.ENQUEUE(
      queue_name          => 'test.multiconsumer_queue',
      enqueue_options     => enqueue_options,
      message_properties  => message_properties,
      payload             => message,
      msgid               => message_handle);

   message := test.message_typ(001, 'MESSAGE 2', 'For two recipients');
   recipients(1) := sys.aq$_agent('RED', NULL, NULL);
   recipients(2) := sys.aq$_agent('BLUE', NULL, NULL);
   message_properties.recipient_list := recipients;
   DBMS_AQ.ENQUEUE(
      queue_name          => 'test.multiconsumer_queue',
      enqueue_options     => enqueue_options,
      message_properties  => message_properties,
      payload             => message,
      msgid               => message_handle);
   COMMIT;
END;
/
```

Example 7-8 Enqueuing Grouped Messages

```
DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             test.message_typ;
BEGIN
  FOR groupno in 1..3 LOOP
    FOR msgno in 1..3 LOOP
      message := test.message_typ(
                001,
                'GROUP ' || groupno,
                'Message ' || msgno || ' in group ' || groupno);
      DBMS_AQ.ENQUEUE(
         queue_name             => 'test.group_queue',
         enqueue_options        => enqueue_options,
         message_properties     => message_properties,
         payload                => message,
         msgid                  => message_handle);
    END LOOP;
    COMMIT;
  END LOOP;
END;
/
```

Example 7-9 Enqueuing a Message, Specifying Delay and Expiration

```
DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             test.message_typ;
BEGIN
   message := test.message_typ(001, 'DELAYED', 'Message is delayed one week.');
   message_properties.delay      := 7*24*60*60;
   message_properties.expiration := 2*7*24*60*60;
   DBMS_AQ.ENQUEUE(
      queue_name           => 'test.obj_queue',
      enqueue_options      => enqueue_options,
      message_properties   => message_properties,
      payload              => message,
      msgid                => message_handle);
   COMMIT;
END;
/
```

Example 7-10 Enqueuing a Message, Specifying a Transformation

```
DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             test.message_typ;
BEGIN
   message := test.message_typ(001, 'NORMAL MESSAGE', 'enqueued to obj_queue');
   enqueue_options.transformation := 'message_order_transform';
   DBMS_AQ.ENQUEUE(
      queue_name              => 'test.priority_queue',
      enqueue_options         => enqueue_options,
      message_properties      => message_properties,
      payload                 => message,
      msgid                   => message_handle);
   COMMIT;
END;
/
```

Enqueuing an Array of Messages

Use the `ENQUEUE_ARRAY` function to enqueue an array of payloads using a corresponding array of message properties.

```
DBMS_AQ.ENQUEUE_ARRAY(
   queue_name                IN   VARCHAR2,
   enqueue_options           IN   enqueue_options_t,
   array_size                IN   PLS_INTEGER,
   message_properties_array  IN   message_properties_array_t,
   payload_array             IN   VARRAY,
   msid_array                OUT  msgid_array_t)
RETURN PLS_INTEGER;
```

The output is an array of message identifiers of the enqueued messages. The function returns the number of messages successfully enqueued.

Array enqueuing is not supported for buffered messages, but you can still use `DBMS_AQ.ENQUEUE_ARRAY()` to enqueue buffered messages by setting `array_size` to `1`.

The `message_properties_array` parameter is an array of message properties. Each element in the payload array must have a corresponding element in this record. All messages in an array have the same delivery mode.

The payload structure can be a VARRAY or nested table. The message IDs are returned into an array of RAW(16) entries of type `DBMS_AQ.msgid_array_t`.

As with array operations in the relational world, it is not possible to provide a single optimum array size that will be correct in all circumstances. Application developers must experiment with different array sizes to determine the optimal value for their particular applications.

> **See Also:**
>
> - "Enqueue Options"
> - "Message Properties"

Example 7-11 Enqueuing an Array of Messages

```
DECLARE
  enqueue_options       DBMS_AQ.enqueue_options_t;
  msg_prop_array        DBMS_AQ.message_properties_array_t;
  msg_prop              DBMS_AQ.message_properties_t;
  payload_array         test.msg_table;
  msgid_array           DBMS_AQ.msgid_array_t;
  retval                PLS_INTEGER;
BEGIN
  payload_array  := msg_table(
      message_typ(001, 'MESSAGE  1', 'array enqueued to obj_queue'),
      message_typ(001, 'MESSAGE  2', 'array enqueued to obj_queue'));
  msg_prop_array := DBMS_AQ.message_properties_array_t(msg_prop, msg_prop);

  retval := DBMS_AQ.ENQUEUE_ARRAY(
                queue_name              => 'test.obj_queue',
                enqueue_options         => enqueue_options,
                array_size              => 2,
                message_properties_array => msg_prop_array,
                payload_array           => payload_array,
                msgid_array             => msgid_array);
  COMMIT;
END;/
```

Listening to One or More Queues

This procedure specifies which queue or queues to monitor.

```
DBMS_AQ.LISTEN(
   agent_list            IN    aq$_agent_list_t,
   wait                  IN    BINARY_INTEGER DEFAULT FOREVER,
   listen_delivery_mode  IN    PLS_INTEGER DEFAULT PERSISTENT,
   agent                 OUT   sys.aq$_agent
   message_delivery_mode OUT   PLS_INTEGER);

TYPE aq$_agent_list_t IS TABLE of aq$_agent INDEXED BY BINARY_INTEGER;
```

This call takes a list of agents as an argument. Each agent is identified by a unique combination of name, address, and protocol.

You specify the queue to be monitored in the address field of each agent listed. Agents must have dequeue privileges on each monitored queue. You must specify the name of the agent when monitoring multiconsumer queues; but you must not specify an agent name for single-consumer queues. Only local queues are supported as addresses. Protocol is reserved for future use.

> **Note:**
>
> Listening to multiconsumer queues is not supported in the Java API.

The `listen_delivery_mode` parameter specifies what types of message interest the agent. If it is the default `PERSISTENT`, then the agent is informed about persistent messages only. If it is set to `BUFFERED`, then the agent is informed about buffered messages only. If it is set to `PERSISTENT_OR_BUFFERED`, then the agent is informed about both types.

This is a blocking call that returns the agent and message type when there is a message ready for consumption for an agent in the list. If there are messages for more than one agent, then only the first agent listed is returned. If there are no messages found when the wait time expires, then an error is raised.

A successful return from the `listen` call is only an indication that there is a message for one of the listed agents in one of the specified queues. The interested agent must still dequeue the relevant message.

> **Note:**
>
> You cannot call `LISTEN` on nonpersistent queues.

Even though both `test.obj_queue` and `test.priority_queue` contain messages (enqueued in Example 7-1 and Example 7-2 respectively) Example 7-12 returns only:

```
Message in Queue: "TEST"."OBJ_QUEUE"
```

If the order of agents in `test_agent_list` is reversed, so `test.priority_queue` appears before `test.obj_queue`, then the example returns:

```
Message in Queue: "TEST"."PRIORITY_QUEUE"
```

> **See Also:**
>
> "AQ Agent Type"

Example 7-12 Listening to a Single-Consumer Queue with Zero Timeout

```
SET SERVEROUTPUT ON
DECLARE
   agent            sys.aq$_agent;
   test_agent_list  DBMS_AQ.aq$_agent_list_t;
BEGIN
   test_agent_list(1) := sys.aq$_agent(NULL, 'test.obj_queue',  NULL);
```

```
   test_agent_list(2) := sys.aq$_agent(NULL, 'test.priority_queue', NULL);
   DBMS_AQ.LISTEN(
      agent_list   =>   test_agent_list,
      wait         =>   0,
      agent        =>   agent);
   DBMS_OUTPUT.PUT_LINE('Message in Queue: ' ||  agent.address);
END;
/
```

Dequeuing Messages

This procedure dequeues a message from the specified queue.

```
DBMS_AQ.DEQUEUE(
   queue_name          IN      VARCHAR2,
   dequeue_options     IN      dequeue_options_t,
   message_properties  OUT     message_properties_t,
   payload             OUT     "type_name",
   msgid               OUT     RAW);
```

You can choose to dequeue only persistent messages, only buffered messages, or both. See `delivery_mode` in the following list of dequeue options.

> **See Also:**
>
> "Message Properties"

Dequeue Options

The `dequeue_options` parameter specifies the options available for the dequeue operation. It has the following attributes:

- `consumer_name`

  A consumer can dequeue a message from a queue by supplying the name that was used in the `AQ$_AGENT` type of the `DBMS_AQADM.ADD_SUBSCRIBER` procedure or the recipient list of the message properties. If a value is specified, then only those messages matching `consumer_name` are accessed. If a queue is not set up for multiple consumers, then this field must be set to NULL (the default).

- `dequeue_mode`

  The `dequeue_mode` attribute specifies the locking behavior associated with the dequeue. If `BROWSE` is specified, then the message is dequeued without acquiring any lock. If `LOCKED` is specified, then the message is dequeued with a write lock that lasts for the duration of the transaction. If `REMOVE` is specified, then the message is dequeued and deleted (the default). The message can be retained in the queue table based on the retention properties. If `REMOVE_NO_DATA` is specified, then the message is marked as updated or deleted.

- `navigation`

  The `navigation` attribute specifies the position of the dequeued message. If `FIRST_MESSAGE` is specified, then the first available message matching the search criteria is dequeued. If `NEXT_MESSAGE` is specified, then the next available message matching the search criteria is dequeued (the default). If the previous message belongs to a message group, then the next available message matching the search criteria in the message group is dequeued.

If NEXT_TRANSACTION is specified, then any messages in the current transaction group are skipped and the first message of the next transaction group is dequeued. This setting can only be used if message grouping is enabled for the queue.

- visibility

  The visibility attribute specifies when the new message is dequeued. If ON_COMMIT is specified, then the dequeue is part of the current transaction (the default). If IMMEDIATE is specified, then the dequeue operation is an autonomous transaction that commits at the end of the operation. The visibility attribute is ignored in BROWSE dequeue mode.

  Visibility must always be IMMEDIATE when dequeuing messages with delivery mode DBMS_AQ.BUFFERED or DBMS_AQ.PERSISTENT_OR_BUFFERED.

- wait

  The wait attribute specifies the wait time if there is currently no message available matching the search criteria. If a number is specified, then the operation waits that number of seconds. If FOREVER is specified, then the operation waits forever (the default). If NO_WAIT is specified, then the operation does not wait.

- msgid

  The msgid attribute specifies the message identifier of the dequeued message. Only messages in the READY state are dequeued unless msgid is specified.

- correlation

  The correlation attribute specifies the correlation identifier of the dequeued message. The correlation identifier cannot be changed between successive dequeue calls without specifying the FIRST_MESSAGE navigation option.

  Correlation identifiers are application-defined identifiers that are not interpreted by Oracle Database Advanced Queuing. You can use special pattern matching characters, such as the percent sign and the underscore. If more than one message satisfies the pattern, then the order of dequeuing is indeterminate, and the sort order of the queue is not honored.

  > **Note:**
  >
  > Although dequeue options correlation and deq_condition are both supported for buffered messages, it is not possible to create indexes to optimize these queries.

- deq_condition

  The deq_condition attribute is a Boolean expression similar to the WHERE clause of a SQL query. This Boolean expression can include conditions on message properties, user data properties (object payloads only), and PL/SQL or SQL functions.

  To specify dequeue conditions on a message payload (object payload), use attributes of the object type in clauses. You must prefix each attribute with tab.user_data as a qualifier to indicate the specific column of the queue table that stores the payload.

  The deq_condition attribute cannot exceed 4000 characters. If more than one message satisfies the dequeue condition, then the order of dequeuing is indeterminate, and the sort order of the queue is not honored.

- transformation

The `transformation` attribute specifies a transformation that will be applied after the message is dequeued but before returning the message to the caller.

- `delivery_mode`

  The `delivery_mode` attribute specifies what types of messages to dequeue. If it is set to `DBMS_AQ.PERSISTENT`, then only persistent messages are dequeued. If it is set to `DBMS_AQ.BUFFERED`, then only buffered messages are dequeued.

  If it is the default `DBMS_AQ.PERSISTENT_OR_BUFFERED`, then both persistent and buffered messages are dequeued. The `delivery_mode` attribute in the message properties of the dequeued message indicates whether the dequeued message was buffered or persistent.

The dequeue order is determined by the values specified at the time the queue table is created unless overridden by the message identifier and correlation identifier in dequeue options.

The database consistent read mechanism is applicable for queue operations. For example, a `BROWSE` call may not see a message that is enqueued after the beginning of the browsing transaction.

In a commit-time queue, messages are not visible to `BROWSE` or `DEQUEUE` calls until a deterministic order can be established among them based on an approximate CSCN.

If the `navigation` attribute of the `dequeue_conditions` parameter is `NEXT_MESSAGE` (the default), then subsequent dequeues retrieve messages from the queue based on the snapshot obtained in the first dequeue. A message enqueued after the first dequeue command, therefore, will be processed only after processing all remaining messages in the queue. This is not a problem if all the messages have already been enqueued or if the queue does not have priority-based ordering. But if an application must process the highest-priority message in the queue, then it must use the `FIRST_MESSAGE` navigation option.

> **Note:**
>
> It can also be more efficient to use the `FIRST_MESSAGE` navigation option when there are messages being concurrently enqueued. If the `FIRST_MESSAGE` option is not specified, then Oracle Database Advanced Queuing continually generates the snapshot as of the first dequeue command, leading to poor performance. If the `FIRST_MESSAGE` option is specified, then Oracle Database Advanced Queuing uses a new snapshot for every dequeue command.

Messages enqueued in the same transaction into a queue that has been enabled for message grouping form a group. If only one message is enqueued in the transaction, then this effectively forms a group of one message. There is no upper limit to the number of messages that can be grouped in a single transaction.

In queues that have not been enabled for message grouping, a dequeue in `LOCKED` or `REMOVE` mode locks only a single message. By contrast, a dequeue operation that seeks to dequeue a message that is part of a group locks the entire group. This is useful when all the messages in a group must be processed as a unit.

When all the messages in a group have been dequeued, the dequeue returns an error indicating that all messages in the group have been processed. The application can then use `NEXT_TRANSACTION` to start dequeuing messages from the next available group. In the event that no groups are available, the dequeue times out after the period specified in the `wait` attribute of `dequeue_options`.

Typically, you expect the consumer of messages to access messages using the dequeue interface. You can view processed messages or messages still to be processed by browsing by message ID or by using `SELECT` commands.

Example 7-13 returns the message enqueued in Example 7-1. It returns:

```
From Sender No.1
Subject: TEST MESSAGE
Text: First message to obj_queue
```

> **See Also:**
>
> - "Dequeue Modes"

Dequeuing LOB Type Messages

Example 7-14 creates procedure `blobdequeue()` to dequeue the LOB type messages enqueued in Example 7-4. The actual dequeue is shown in Example 7-15. It returns:

```
Amount of data read: 2000
Amount of data read: 2000
Amount of data read: 2000
Amount of data read: 2000
Amount of data read: 2000
```

Dequeuing Grouped Messages

You can dequeue the grouped messages enqueued in Example 7-8 by running Example 7-16. It returns:

```
GROUP 1: Message 1 in group 1
GROUP 1: Message 2 in group 1
GROUP 1: Message 3 in group 1
Finished GROUP 1
GROUP 2: Message 1 in group 2
GROUP 2: Message 2 in group 2
GROUP 2: Message 3 in group 2
Finished GROUP 2
GROUP 3: Message 1 in group 3
GROUP 3: Message 2 in group 3
GROUP 3: Message 3 in group 3
Finished GROUP 3
No more messages
```

Dequeuing from a Multiconsumer Queue

You can dequeue the messages enqueued for `RED` in Example 7-7 by running Example 7-17. If you change `RED` to `GREEN` and then to `BLUE`, you can use it to dequeue their messages as well. The output of the example will be different in each case.

`RED` is a subscriber to the multiconsumer queue and is also a specified recipient of MESSAGE 2, so it gets both messages:

```
Message: MESSAGE 1 .. For queue subscribers
Message: MESSAGE 2 .. For two recipients
No more messages for RED
```

GREEN is only a subscriber, so it gets only those messages in the queue for which no recipients have been specified (in this case, MESSAGE 1):

```
Message: MESSAGE 1 .. For queue subscribers
No more messages for GREEN
```

BLUE, while not a subscriber to the queue, is nevertheless specified to receive MESSAGE 2.

```
Message: MESSAGE 2 .. For two recipients
No more messages for BLUE
```

Example 7-18 browses messages enqueued in Example 7-5 until it finds PINK, which it removes. The example returns:

```
Browsed Message Text: ORANGE enqueued first.
Browsed Message Text: ORANGE also enqueued second.
Browsed Message Text: YELLOW enqueued third.
Browsed Message Text: VIOLET enqueued fourth.
Browsed Message Text: PURPLE enqueued fifth.
Browsed Message Text: PINK enqueued sixth.
Removed Message Text: PINK enqueued sixth.
```

Dequeue Modes

Example 7-19 previews in locked mode the messages enqueued in Example 7-5 until it finds PURPLE, which it removes. The example returns:

```
Locked Message Text: ORANGE enqueued first.
Locked Message Text: ORANGE also enqueued second.
Locked Message Text: YELLOW enqueued third.
Locked Message Text: VIOLET enqueued fourth.
Locked Message Text: PURPLE enqueued fifth.
Removed Message Text: PURPLE enqueued fifth.
```

Example 7-13 Dequeuing Object Type Messages

```
SET SERVEROUTPUT ON
DECLARE
dequeue_options     DBMS_AQ.dequeue_options_t;
message_properties  DBMS_AQ.message_properties_t;
message_handle      RAW(16);
message             test.message_typ;
BEGIN
   dequeue_options.navigation := DBMS_AQ.FIRST_MESSAGE;
   DBMS_AQ.DEQUEUE(
      queue_name          =>     'test.obj_queue',
      dequeue_options     =>     dequeue_options,
      message_properties  =>     message_properties,
      payload             =>     message,
      msgid               =>     message_handle);
   DBMS_OUTPUT.PUT_LINE('From Sender No.'|| message.sender_id);
   DBMS_OUTPUT.PUT_LINE('Subject: '||message.subject);
   DBMS_OUTPUT.PUT_LINE('Text: '||message.text);
   COMMIT;
END;
/
```

Example 7-14 Creating a Dequeue Procedure for LOB Type Messages

```
CREATE OR REPLACE PROCEDURE blobdequeue(msgno IN NUMBER) AS
   dequeue_options     DBMS_AQ.dequeue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   msgid               RAW(16);
```

```
   payload             test.lob_typ;
   lob_loc             BLOB;
   amount              BINARY_INTEGER;
   buffer              RAW(4096);
BEGIN
   DBMS_AQ.DEQUEUE(
      queue_name          =>  'test.lob_queue',
      dequeue_options     =>   dequeue_options,
      message_properties  =>   message_properties,
      payload             =>   payload,
      msgid               =>   msgid);
   lob_loc                :=   payload.data;
   amount                 :=   2000;
   DBMS_LOB.READ(lob_loc, amount, 1, buffer);
   DBMS_OUTPUT.PUT_LINE('Amount of data read: '|| amount);
   COMMIT;
END;
/
```

Example 7-15 Dequeuing LOB Type Messages

```
BEGIN
   FOR i IN 1..5 LOOP
     blobdequeue(i);
   END LOOP;
END;
/
```

Example 7-16 Dequeuing Grouped Messages

```
SET SERVEROUTPUT ON
DECLARE
   dequeue_options       DBMS_AQ.dequeue_options_t;
   message_properties    DBMS_AQ.message_properties_t;
   message_handle        RAW(16);
   message               test.message_typ;
   no_messages           exception;
   end_of_group          exception;
   PRAGMA EXCEPTION_INIT (no_messages, -25228);
   PRAGMA EXCEPTION_INIT (end_of_group, -25235);
BEGIN
   dequeue_options.wait       := DBMS_AQ.NO_WAIT;
   dequeue_options.navigation := DBMS_AQ.FIRST_MESSAGE;
   LOOP
     BEGIN
     DBMS_AQ.DEQUEUE(
        queue_name         => 'test.group_queue',
        dequeue_options    => dequeue_options,
        message_properties => message_properties,
        payload            => message,
        msgid              => message_handle);
     DBMS_OUTPUT.PUT_LINE(message.subject || ': ' || message.text );
     dequeue_options.navigation := DBMS_AQ.NEXT_MESSAGE;
     EXCEPTION
       WHEN end_of_group THEN
         DBMS_OUTPUT.PUT_LINE ('Finished ' || message.subject);
         COMMIT;
         dequeue_options.navigation := DBMS_AQ.NEXT_TRANSACTION;
     END;
   END LOOP;
   EXCEPTION
     WHEN no_messages THEN
       DBMS_OUTPUT.PUT_LINE ('No more messages');
```

```
END;
/
```

Example 7-17 Dequeuing Messages for RED from a Multiconsumer Queue

```
SET SERVEROUTPUT ON
DECLARE
  dequeue_options       DBMS_AQ.dequeue_options_t;
  message_properties    DBMS_AQ.message_properties_t;
  message_handle        RAW(16);
  message               test.message_typ;
  no_messages           exception;
  PRAGMA EXCEPTION_INIT (no_messages, -25228);
BEGIN
  dequeue_options.wait          := DBMS_AQ.NO_WAIT;
  dequeue_options.consumer_name := 'RED';
  dequeue_options.navigation    := DBMS_AQ.FIRST_MESSAGE;
  LOOP
   BEGIN
    DBMS_AQ.DEQUEUE(
      queue_name         => 'test.multiconsumer_queue',
      dequeue_options    => dequeue_options,
      message_properties => message_properties,
      payload            => message,
      msgid              => message_handle);
    DBMS_OUTPUT.PUT_LINE('Message: '|| message.subject ||' .. '|| message.text );
    dequeue_options.navigation := DBMS_AQ.NEXT_MESSAGE;
   END;
  END LOOP;
  EXCEPTION
    WHEN no_messages THEN
    DBMS_OUTPUT.PUT_LINE ('No more messages for RED');
  COMMIT;
END;
/
```

Example 7-18 Dequeue in Browse Mode and Remove Specified Message

```
SET SERVEROUTPUT ON
DECLARE
   dequeue_options     DBMS_AQ.dequeue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             test.message_typ;
BEGIN
   dequeue_options.dequeue_mode := DBMS_AQ.BROWSE;
   LOOP
      DBMS_AQ.DEQUEUE(
        queue_name              => 'test.obj_queue',
        dequeue_options         => dequeue_options,
        message_properties      => message_properties,
        payload                 => message,
        msgid                   => message_handle);
      DBMS_OUTPUT.PUT_LINE ('Browsed Message Text: ' || message.text);
      EXIT WHEN message.subject = 'PINK';
   END LOOP;
   dequeue_options.dequeue_mode := DBMS_AQ.REMOVE;
   dequeue_options.msgid        := message_handle;
   DBMS_AQ.DEQUEUE(
           queue_name           => 'test.obj_queue',
           dequeue_options      => dequeue_options,
           message_properties   => message_properties,
           payload              => message,
```

```
          msgid               => message_handle);
   DBMS_OUTPUT.PUT_LINE('Removed Message Text: ' || message.text);
   COMMIT;
END;
/
```

Example 7-19 Dequeue in Locked Mode and Remove Specified Message

```
SET SERVEROUTPUT ON
DECLARE
   dequeue_options     DBMS_AQ.dequeue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             test.message_typ;
BEGIN
   dequeue_options.dequeue_mode := DBMS_AQ.LOCKED;
   LOOP
      DBMS_AQ.dequeue(
        queue_name         => 'test.obj_queue',
        dequeue_options    => dequeue_options,
        message_properties => message_properties,
        payload            => message,
        msgid              => message_handle);
      DBMS_OUTPUT.PUT_LINE('Locked Message Text: ' || message.text);
      EXIT WHEN message.subject = 'PURPLE';
   END LOOP;
   dequeue_options.dequeue_mode := DBMS_AQ.REMOVE;
   dequeue_options.msgid        := message_handle;
   DBMS_AQ.DEQUEUE(
     queue_name           => 'test.obj_queue',
     dequeue_options      => dequeue_options,
     message_properties   => message_properties,
     payload              => message,
     msgid                => message_handle);
   DBMS_OUTPUT.PUT_LINE('Removed Message Text: ' || message.text);
   COMMIT;
END;
/
```

Dequeuing an Array of Messages

Use the `DEQUEUE_ARRAY` function to dequeue an array of payloads and a corresponding array of message properties.

```
DBMS_AQ.DEQUEUE_ARRAY(
   queue_name                IN      VARCHAR2,
   dequeue_options           IN      dequeue_options_t,
   array_size                IN      PLS_INTEGER,
   message_properties_array  OUT     message_properties_array_t,
   payload_array             OUT     VARRAY,
   msgid_array               OUT     msgid_array_t)
RETURN PLS_INTEGER;
```

The output is an array of payloads, message IDs, and message properties of the dequeued messages. The function returns the number of messages successfully dequeued.

Array dequeuing is not supported for buffered messages, but you can still use `DBMS_AQ.DEQUEUE_ARRAY()` to dequeue buffered messages by setting `array_size` to `1`.

The payload structure can be a `VARRAY` or nested table. The message identifiers are returned into an array of `RAW(16)` entries of type `DBMS_AQ.msgid_array_t`. The message properties are returned into an array of type `DBMS_AQ.message_properties_array_t`.

As with array operations in the relational world, it is not possible to provide a single optimum array size that will be correct in all circumstances. Application developers must experiment with different array sizes to determine the optimal value for their particular applications.

All dequeue options available with `DBMS_AQ.DEQUEUE` are also available with `DBMS_AQ.DEQUEUE_ARRAY`. You can choose to dequeue only persistent messages, only buffered messages, or both. In addition, the `navigation` attribute of `dequeue_options` offers two options specific to `DBMS_AQ.DEQUEUE_ARRAY`.

When dequeuing messages, you might want to dequeue all the messages for a transaction group with a single call. You might also want to dequeue messages that span multiple transaction groups. You can specify either of these methods by using one of the following navigation methods:

- `NEXT_MESSAGE_ONE_GROUP`
- `FIRST_MESSAGE_ONE_GROUP`
- `NEXT_MESSAGE_MULTI_GROUP`
- `FIRST_MESSAGE_MULTI_GROUP`

Navigation method `NEXT_MESSAGE_ONE_GROUP` dequeues messages that match the search criteria from the next available transaction group into an array. Navigation method `FIRST_MESSAGE_ONE_GROUP` resets the position to the beginning of the queue and dequeues all the messages in a single transaction group that are available and match the search criteria.

The number of messages dequeued is determined by an array size limit. If the number of messages in the transaction group exceeds `array_size`, then multiple calls to `DEQUEUE_ARRAY` must be made to dequeue all the messages for the transaction group.

Navigation methods `NEXT_MESSAGE_MULTI_GROUP` and `FIRST_MESSAGE_MULTI_GROUP` work like their `ONE_GROUP` counterparts, but they are not limited to a single transaction group. Each message that is dequeued into the array has an associated set of message properties. Message property `transaction_group` determines which messages belong to the same transaction group.

Example 7-20 dequeues the messages enqueued in Example 7-11. It returns:

```
Number of messages dequeued: 2
```

> **See Also:**
>
> "Dequeuing Messages"

Example 7-20 Dequeuing an Array of Messages

```
SET SERVEROUTPUT ON
DECLARE
  dequeue_options       DBMS_AQ.dequeue_options_t;
  msg_prop_array        DBMS_AQ.message_properties_array_t :=
                        DBMS_AQ.message_properties_array_t();
  payload_array         test.msg_table;
  msgid_array           DBMS_AQ.msgid_array_t;
```

```
  retval               PLS_INTEGER;
BEGIN
  retval := DBMS_AQ.DEQUEUE_ARRAY(
              queue_name              => 'test.obj_queue',
              dequeue_options         => dequeue_options,
              array_size              => 2,
              message_properties_array => msg_prop_array,
              payload_array           => payload_array,
              msgid_array             => msgid_array);
  DBMS_OUTPUT.PUT_LINE('Number of messages dequeued: ' || retval);
END;/
```

Registering for Notification

This procedure registers an e-mail address, user-defined PL/SQL procedure, or HTTP URL for message notification.

```
DBMS_AQ.REGISTER(
   reg_list     IN SYS.AQ$_REG_INFO_LIST,
   reg_count    IN NUMBER);
```

> **Note:**
>
> Starting from 12*c* Release 2 (12.2.), the maximum length of user-generated queue names is 122 bytes. See "Creating a Queue".

The `reg_list` parameter is a list of `SYS.AQ$_REG_INFO` objects. You can specify notification quality of service with the `qosflags` attribute of `SYS.AQ$_REG_INFO`.

The `reg_count` parameter specifies the number of entries in the `reg_list`. Each subscription requires its own `reg_list` entry. Interest in several subscriptions can be registered at one time.

When PL/SQL notification is received, the Oracle Database Advanced Queuing message properties descriptor that the callback is invoked with specifies the `delivery_mode` of the message notified as `DBMS_AQ.PERSISTENT` or `DBMS_AQ.BUFFERED`.

If you register for e-mail notifications, then you must set the host name and port name for the SMTP server that will be used by the database to send e-mail notifications. If required, you should set the send-from e-mail address, which is set by the database as the `sent from` field. You need a Java-enabled database to use this feature.

If you register for HTTP notifications, then you might want to set the host name and port number for the proxy server and a list of no-proxy domains that will be used by the database to post HTTP notifications.

An internal queue called `SYS.AQ_SRVNTFN_TABLE_Q` stores the notifications to be processed by the job queue processes. If notification fails, then Oracle Database Advanced Queuing retries the failed notification up to `MAX_RETRIES` attempts.

> **Note:**
>
> You can change the `MAX_RETRIES` and `RETRY_DELAY` properties of `SYS.AQ_SRVNTFN_TABLE_Q`. The new settings are applied across all notifications.

> **See Also:**
>
> - "AQ Registration Information Type" for more information on `SYS.AQ$_REG_INFO` objects
> - "AQ Notification Descriptor Type" for more information on the message properties descriptor

Example 7-21 Registering for Notifications

```
DECLARE
  reginfo             sys.aq$_reg_info;
  reg_list            sys.aq$_reg_info_list;
BEGIN
  reginfo := sys.aq$_reg_info(
                      'test.obj_queue',
                      DBMS_AQ.NAMESPACE_ANONYMOUS,
                      'http://www.company.com:8080',
                      HEXTORAW('FF'));
  reg_list  := sys.aq$_reg_info_list(reginfo);
  DBMS_AQ.REGISTER(
    reg_list     => reg_list,
    reg_count    => 1);
  COMMIT;
END;
/
```

Unregistering for Notification

This procedure unregisters an e-mail address, user-defined PL/SQL procedure, or HTTP URL for message notification.

```
DBMS_AQ.UNREGISTER(
   reg_list     IN SYS.AQ$_REG_INFO_LIST,
   reg_count    IN NUMBER);
```

Posting for Subscriber Notification

This procedure posts to a list of anonymous subscriptions, allowing all clients who are registered for the subscriptions to get notifications of persistent messages.

```
DBMS_AQ.POST(
  post_list       IN  SYS.AQ$_POST_INFO_LIST,
  post_count      IN  NUMBER);
```

This feature is not supported with buffered messages.

The `count` parameter specifies the number of entries in the `post_list`. Each posted subscription must have its own entry in the `post_list`. Several subscriptions can be posted to at one time.

The `post_list` parameter specifies the list of anonymous subscriptions to which you want to post. It has three attributes:

- `name`

  The `name` attribute specifies the name of the anonymous subscription to which you want to post.

- `namespace`

  The `namespace` attribute specifies the namespace of the subscription. To receive notifications from other applications through `DBMS_AQ.POST` the namespace must be `DBMS_AQ.NAMESPACE_ANONYMOUS`.

- `payload`

  The `payload` attribute specifies the payload to be posted to the anonymous subscription. It is possible for no payload to be associated with this call.

This call provides a best-effort guarantee. A notification goes to registered clients at most once. This call is primarily used for lightweight notification. If an application needs more rigid guarantees, then it can enqueue to a queue.

Example 7-22 Posting Object-Type Messages

```
DECLARE
  postinfo            sys.aq$_post_info;
  post_list           sys.aq$_post_info_list;
BEGIN
  postinfo  := sys.aq$_post_info('test.obj_queue',0,HEXTORAW('FF'));
  post_list := sys.aq$_post_info_list(postinfo);
  DBMS_AQ.POST(
    post_list       => post_list,
    post_count      => 1);
  COMMIT;
END;
/
```

Adding an Agent to the LDAP Server

This procedure creates an entry for an Oracle Database Advanced Queuing agent in the LDAP server.

```
DBMS_AQ.BIND_AGENT(
   agent        IN SYS.AQ$_AGENT,
   certificate  IN VARCHAR2 default NULL);
```

The `agent` parameter specifies the Oracle Database Advanced Queuing Agent that is to be registered in Lightweight Directory Access Protocol (LDAP) server.

The `certificate` parameter specifies the location (LDAP distinguished name) of the `OrganizationalPerson` entry in LDAP whose digital certificate (attribute `usercertificate`) is to be used for this agent. For example, "`cn=OE, cn=ACME, cn=com`" is a distinguished name for a `OrganizationalPerson OE` whose certificate will be used with the specified agent. If the agent does not have a digital certificate, then this parameter is defaulted to null.

> **See Also:**
>
> "AQ Agent Type"

Removing an Agent from the LDAP Server

This procedure removes the entry for an Oracle Database Advanced Queuing agent from the LDAP server.

```
DBMS_AQ.UNBIND_AGENT(
   agent     IN SYS.AQ$_AGENT);
```