Oracle Advanced Queuing (AQ)

A customer asked me to do a presentation about AQ (Advanced Queuing). This article is a rewrite of that presentation. We'll be getting some hands-on experience with AQ, and then some tips on the issues that might occur.
As always, your comments to this article are more than welcome. If you enjoy this article, find it useful or maybe not at all, please let me know by leaving a comment on the site.

Queuing: I don't want to queue.

The title might be true in many cases, but there are also situations where a queue is very convenient. For example in the case of batch processing where a batch process handles multiple incoming messages from an online process. Or when 2 processes need inter-process communication, but still need to function independently of each other.

In eBS we use queues for the workflow system. (Deferred items, notifications for the workflow mailer and the Business Event System). Some more queues are found for concurrent processing and SFM.

So whether you like it or not, you'll have to queue. The trick is to manage these queues to get optimal performance for your system.

(Advanced) Queue design

Before we can start building queues, there are some things to consider.
AQ supports both point-to-point queues and publish-subscribe-queues (also called multi-consumer queues).

Point-to-point queues are emptied by only one specific process. One or more processes can enqueue messages on the queue, but only one process can dequeue them.

In contrast, a publish-subscribe queue can have many processes reading the messages in the queue. Either the messages are broadcasted, or the receivers have to subscribe to a certain kind of messages. Of course the publish-subscribe queue has some very interesting properties. But we'll start our item with the point-to-point queue.

So you'll first have to decide who the senders and receivers of the queue data will be. In this article, we start with using a point-to-point queue. After that we start using multi-consumer queues.

Another thing to consider is the payload of the message. Of course, the messages will need some content to give it a meaning to the receiver. This content is called the payload. And you can either use a custom type (including XML), or a raw type.

During this article, we'll see some more features of AQ. But when we decide on the type of queue and the payload type, we can build our own queues. All queues are built on queue-tables. These tables hold the data in the queue. On top of these tables, the actual queue and some management views are built.

To build a queue-table, we use the dbms_aqadm package:

```
dbms_aqadm.create_queue_table(queue_table =>'<table_name>'
                              ,queue_payload_type => ['RAW'|<custom_type>]);
```

This creates the queue table including a LOB segment for the payload, some indexes, and an 'Error queue':

```
Begin
dbms_aqadm.create_queue_table(queue_table=>'xxx_test'
                               ,queue_payload_type=>'RAW');
End;


Select object_name,object_type from dba_objects where created>sysdate-1/24;

OBJECT_NAME                    OBJECT_TYPE
------------------------------ -----------
SYS_C0011768                   INDEX
XXX_TEST                       TABLE
SYS_LOB0000073754C00029$$      LOB
SYS_LOB0000073754C00028$$      LOB
AQ$_XXX_TEST_T                 INDEX
AQ$_XXX_TEST_I                 INDEX
AQ$_XXX_TEST_E                 QUEUE
AQ$_XXX_TEST_F                 VIEW
AQ$XXX_TEST                    VIEW
```

This created the base-table for a point-to-point queue. The table is a regular heap-oriented table. And you are free to create extra indexes on it, if you feel the urge. The necessary indexes have been created already.

The queue that is created now is the default error queue. Messages that failed dequeuing will be set on this queue.

Now it's time to create the actual queue. The queue-tables are the infrastructure for storing the messages and related information. The queue can now be created to control the queuing and dequeuing of messages.

For both point-to-point as publish-subscriber queues, the command is:

```
dbms_aqadm.create_queue (queue_table =>'<table_name>'
                         queue_name => '<queue_name>');
```

So for us we run:

```
Begin
dbms_aqadm.create_queue (queue_name => 'xxx_test_q'
                         ,queue_table => 'xxx_mc_test');
End;
```

This creates an object of type QUEUE. This is the object that will control the contents of the underlying tables / IOT's.

Before we can start using our queues, we also have to 'start' them. On starting, we indicate whether the queue is available for queuing, dequeuing or both:

```
Begin
 dbms_aqadm.start_queue(queue_name=>'xxx_test_q'
                        ,enqueue=>TRUE
                        ,dequeue=>TRUE);
End;
```

Our queue is now enabled for both queueing and dequeuing. Let's first verify if things are working correctly.

To enqueue (or dequeue) a message, we use the dbms_aq package. It has an enqueue and dequeue procedure. Both with their own parameters. The parameters include en-/dequeue options, message properties, a message_id and of course the message itself:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_enq_options Dbms_aq.Enqueue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
V_payload:=Utl_raw.Cast_to_raw('Hello world!');
Dbms_aq.Enqueue(Queue_name=>'xxx_test_q'
                ,Message_properties=>V_msg_properties
                ,Enqueue_options=>V_enq_options
                ,Payload=>V_payload
                ,Msgid=>V_msgid);
Dbms_output.Put_line(rawtohex(V_msgid));
end;
```

This enqueues a 'Hello world!' message, and returns the message id. If everything works correctly, you'll see the msgid as a hexadecimal string. (Don't forget to set serveroutput on).

We created 2 extra parameters: v_enq_options with the options used to enqueue this message. And v_msg_properties to set additional information about the message.
V_enq_options is of type 'dbms_aq.enqueue_options_t'. This is a record of:

```
Visibility     BINARY_INTEGER  --Options are: dbms_aq.on_commit and dbms_aq.immediate. This
       indicates whether the enqueue is part of the current transaction, or done autonomously.
Relative_msgid RAW(16)        --If the message needs to be enqueued at a specific position, it
       will be relative to this msgid.
Sequence_deviation BINARY_INTEGER --Options are: DBMS_AQ.BEFORE, DBMS_AQ.TOP or NULL (default).
       If before then the message is before the relative_msgid. If top, the message will be the
       first to be dequeued.
```

V_msg_properties is of type ‘dbms_aq.message_properties_t’. This is a record of:

```
priority        BINARY_INTEGER  -- Any integer, to set the priority. Smaller is higher priority.
       The default is 1.
delay           BINARY_INTEGER  -- If the message needs to be delayed before it can be dequeued,
       set the time in seconds here. The default is dbms_aq.no_delay.
expiration      BINARY_INTEGER  -- For messages that need to expire after a certain time, set the
       expiration time in seconds. (Offset from the delay). The default is dbms_aq.never.
correlation     VARCHAR2(128)   -- A free text field that can be used to identify groups of
       messages.
attempts        BINARY_INTEGER  -- Number of failed attempts to dequeue, before the message will
       be failed and marked as expired.
recipient_list  DBMS_AQ.AQ$_RECIPIENT_LIST_T -- Only valid for multi-consumer queues. Sets the
       designated recipients.
exception_queue VARCHAR2(51)    -- The exception queue to use, when it is not the default.
enqueue_time    DATE            -- Set automatically during enqueue
state           BINARY_INTEGER  -- Automatically maintained by AQ, to indicate the status of the
       message.
```

Let’s see if the dequeue also works. For this the procedure dequeue is used, with similar parameters.

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
Dbms_aq.dequeue(Queue_name=>'xxx_test_q'
               ,Message_properties=>V_msg_properties
               ,dequeue_options=>V_deq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
End;
```

This time, our message should be displayed.

For the dequeue, we used v_deq_options of type ‘dbms_aq.dequeue_options_t’. This is a record of:

```
consumer_name  VARCHAR2(30)   -- Indicates the consumer for multi-consumer queues.
dequeue_mode   BINARY_INTEGER -- How to dequeue the messages. Either leave it on the queue, or
       remove it. Either dbms_aq.browse and dbms_aq.remove (default).
navigation     BINARY_INTEGER -- Indicate where to start dequeuing. Dbms_aq.next_message
       (default), to continue from the previous dequeue. Dbms_aq.first_message to start at the
       top of the queue. Dbms_aq.next_transaction to skip the rest of this message group.
visibility     BINARY_INTEGER -- same as dbms_aq.enqueue_options_t.visibility.
wait           BINARY_INTEGER -- The time (in seconds) the package should wait if no message is
       available. Default is dbms_aq.forever.
msgid          RAW(16)        -- When specified, only the message with this msgid will be
       dequeued.
correlation    VARCHAR2(128)  -- Only messages with this correlation will be dequeued (may
       include wildcards).
```

Note how message_properties and payload are now out-parameters.

This is probably the simplest queue possible. We enqueued and dequeued a raw message. We didn’t specify the visibility. So your session still needs to commit these actions.

Now let’s enqueue our message again, and see how it works behind the curtain.

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_enq_options Dbms_aq.Enqueue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
V_payload:=Utl_raw.Cast_to_raw('Hello world!');
Dbms_aq.Enqueue(Queue_name=>'xxx_test_q'
                ,Message_properties=>V_msg_properties
                ,Enqueue_options=>V_enq_options
                ,Payload=>V_payload
                ,Msgid=>V_msgid);
Dbms_output.Put_line('Msg_id: '||rawtohex(V_msgid));
end;

Msg_id: 499CE4809F2641E1BFBC8AFBC8DB5AFA
```

The queue table is an ordinary heap-table, so we can query it.

```
select q_name, rawtohex(msgid) msg_id, priority, state, enq_time, enq_uid
from   xxx_test;

q_name      msg_id                          priority state enq_time                    enq_uid
XXX_TEST_Q  499CE4809F2641E1BFBC8AFBC8DB5AFA        1     0 21-03-10 17:24:01,876000000   SYSTEM
```

We see our msgid again. A priority flag. A state flag, the time of enqueueing the message, and the user that enqueued the message. The message is also in the table, but since it is a blob, we won't bother selecting from it yet.

There are more columns in the table, that control the order and by who the messages are dequeued. Most of them are still null, so we will see them when needed.

A useful alternative to the table is to query the queue-view aq$<table_name>. This will show the translated values of the state. (0 = READY). And especially when using multi-consumer queues, it will use a join to select a more complete picture of the queue.

When we dequeue the message, it will disappear from the queue. (And be deleted from the queue table). However, this can be controlled by the retention parameter of the queue.

Let's set this parameter, so we can check the data after the dequeue.
We set the retention time to 240 (seconds):

```
begin
DBMS_AQADM.ALTER_QUEUE(queue_name =>'xxx_test_q'
                      ,retention_time => 240);
end;
```

Now when we dequeue the message, it will remain in the queue:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
```

```
Dbms_aq.dequeue(Queue_name=>'xxx_test_q'
               ,Message_properties=>V_msg_properties
               ,dequeue_options=>V_deq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
End;

select  queue
,       rawtohex(msg_id)msg_id
,       msg_priority
,       msg_state
,       enq_timestamp
,       enq_user_id
,       deq_timestamp
,       deq_user_id
from    aq$xxx_test

QUEU      MSG_ID                           MSG_PRIO MSG_STATE ENQ_TIMESTAMP     ENQ_USER DEQ_TIMESTAMP     DEQ_USER_ID
XXX_TEST_Q AEC2CD2E34514363B6739969E8E8D353        1 PROCESSED 19-03-10 18:31:40 SYSTEM   19-03-10 21:26:45 SYSTEM
```

Now the message has been set to state ‘PROCESSED’, and some dequeue information has been added.

It’s time to start navigating queues when there are multiple messages in the queue.

Messages are by default dequeued in the order in which they are enqueued. On creation of the queue table, you can set other dequeue orders. But it is also possible to dequeue messages in a different order by navigating the queues, or using filter-criteria.

To show the dequeueing order we enqueue 10 different messages.

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_enq_options Dbms_aq.Enqueue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
V_enq_options.visibility := dbms_aq.immediate;
for i in 1..10 loop
    V_payload:=Utl_raw.Cast_to_raw('This is the '||to_char(to_date(i,'J'),'jspth')||' message');
    Dbms_aq.Enqueue(Queue_name=>'xxx_test_q'
                   ,Message_properties=>V_msg_properties
                   ,Enqueue_options=>V_enq_options
                   ,Payload=>V_payload
                   ,Msgid=>V_msgid);
    Dbms_output.Put_line(rawtohex(V_msgid));
end loop;
end;
```

This enqueues the text ‘This is the first message’ till ‘This is the tenth message’. On dequeuing, the messages come out in the same order:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
```

```
for i in 1..10 loop
    Dbms_aq.dequeue(Queue_name=>'xxx_test_q'
                   ,Message_properties=>V_msg_properties
                   ,dequeue_options=>V_deq_options
                    ,Payload=>V_payload
                   ,Msgid=>V_msgid);
    Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
end loop;
End;

This is the first message
This is the second message
……
This is the tenth message
```

When we created the queue table, we choose the default sort order. This is by enqueue_time. We can also build a queue that uses priority dequeuing. First we create a queue:

```
begin
dbms_aqadm.create_queue_table(queue_table=>'xxx_test_prio'
                             ,sort_list => 'PRIORITY,ENQ_TIME'
                             ,queue_payload_type=>'RAW');
dbms_aqadm.create_queue(queue_name=>'xxx_test_prio_q'
                       ,queue_table=>'xxx_test_prio');
dbms_aqadm.start_queue(queue_name=>'xxx_test_prio_q');
end;
```

We indicated a sort_list now. The options are ‘ENQ_TIME’ (default), ‘ENQ_TIME,PRIORITY’,‘PRIORITY’,’PRIORITY,ENQ_TIME‘. Now we enqueue some messages with reversed priorities:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_enq_options Dbms_aq.Enqueue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
V_enq_options.visibility := dbms_aq.immediate;
for i in 1..10 loop
    V_payload:=Utl_raw.Cast_to_raw('This is the '||to_char(to_date(i,'J'),'jspth')||' message');
    v_msg_properties.priority:=11-i;
    Dbms_aq.Enqueue(Queue_name=>'xxx_test_prio_q'
                   ,Message_properties=>V_msg_properties
                   ,Enqueue_options=>V_enq_options
                   ,Payload=>V_payload
                   ,Msgid=>V_msgid);
    Dbms_output.Put_line(rawtohex(V_msgid));
end loop;
end;
```

And we dequeue them again:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
```

```
Begin
for i in 1..10 loop
    Dbms_aq.dequeue(Queue_name=>'xxx_test_prio_q'
                    ,Message_properties=>V_msg_properties
                    ,dequeue_options=>V_deq_options
                    ,Payload=>V_payload
                    ,Msgid=>V_msgid);
    Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
end loop;
End;

This is the tenth message
This is the ninth message
……
This is the second message
This is the first message
```

Now it's time to look at queueing navigation. It is possible to dequeue specific messages from the queue. You can select messages with a specific msg_id, correlation or recipient_list (for mc-queueus).

We'll first search for a specific correlation and then a message_id. We enqueue ten messages, with different correlations:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_enq_options Dbms_aq.Enqueue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
V_enq_options.visibility := dbms_aq.immediate;
for i in 1..10 loop
    V_payload:=Utl_raw.Cast_to_raw('This is the '||to_char(to_date(i,'J'),'jspth')||' message');
    v_msg_properties.correlation:=to_char('Corr'||i);
    Dbms_aq.Enqueue(Queue_name=>'xxx_test_q'
                    ,Message_properties=>V_msg_properties
                    ,Enqueue_options=>V_enq_options
                    ,Payload=>V_payload
                    ,Msgid=>V_msgid);
    dbms_output.Put_line('Msg_id: '||rawtohex(V_msgid)||' Correlation: Corr'||i);
end loop;
end;

Msg_id: E8BE83A2A2A04F1EA74863B4A7C78DAF Correlation: Corr1
Msg_id: 7159B80BC3194C7AAA6910AB10E753C5 Correlation: Corr2
Msg_id: 4AF3693CF7EE4994B0F78830371437B9 Correlation: Corr3
Msg_id: 44DBC0CB09C94BB98DF2D7E48971849C Correlation: Corr4
Msg_id: 98F3E119041E47F5BF46604E014120BF Correlation: Corr5
Msg_id: B71B7F097A9E4EDBA696958326BF6300 Correlation: Corr6
Msg_id: C4F5050B02904EEEAD2842405A0BDE2A Correlation: Corr7
Msg_id: E4D923A4CB4B4DF2B64B8421A88FFC42 Correlation: Corr8
Msg_id: BE199053188648AE8FA238A01A5C9CD1 Correlation: Corr9
Msg_id: 8991E793D2DB41F5B3F9D00D283B6F6D Correlation: Corr10
```

Now we can dequeue the 5th (correlation) and 8th (msg_id) message:

```
Declare
V_payload Raw(200);
```

```
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
v_deq_options.correlation:='Corr5';
Dbms_aq.dequeue(Queue_name=>'xxx_test_q'
               ,Message_properties=>V_msg_properties
               ,dequeue_options=>V_deq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
v_deq_options.correlation:=NULL;
v_deq_options.msgid:='E4D923A4CB4B4DF2B64B8421A88FFC42';
Dbms_aq.dequeue(Queue_name=>'xxx_test_q'
               ,Message_properties=>V_msg_properties
               ,dequeue_options=>V_deq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
End;

This is the fifth message
This is the eighth message
```

Note how we have to set the correlation back to NULL for the second dequeue. Otherwise we would be trying to dequeue a message with correlation 'Corr5' and the specified msg_id. Since that message does not exist, our procedure will just wait for the message to appear.

By default when you dequeue from an empty queue, or try to dequeue a non-available message, the dequeue will wait indefinitely for a message to appear. You can control this behavior with the dequeue options.

```
V_deq_options.wait := 10; -- to wait 10 seconds. Any number of 0 or higher is allowed.
V_deq_options.wait := dbms_aq.no_wait; -- not waiting for the message.
V_deq_options.wait := dbms_aq.forever; -- wait indefinitely
```

Do note that when the time-out is reached an `'ORA-25228: timeout in dequeue from <queue> while waiting for a message'` raised. So you will need to handle the exception.

One more feature to consider is the browsing mode. So far we have seen the messages that we dequeued were removed from the queue (or at least got status 'Processed'). By setting the dequeue options, we can first inspect messages before dequeuing them. Consider the following. We have 8 messages left in our queue:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_deq_options_rm dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
v_deq_options.wait:=dbms_aq.no_wait;
v_deq_options.dequeue_mode:=DBMS_AQ.BROWSE;
for i in 1..10 loop
begin
```

```
   Dbms_aq.dequeue(Queue_name=>'xxx_test_q'
                  ,Message_properties=>V_msg_properties
                  ,dequeue_options=>V_deq_options
                  ,Payload=>V_payload
                  ,Msgid=>V_msgid);
   Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
   dbms_output.put_line(v_msg_properties.correlation);
   if v_msg_properties.correlation='Corr6' then
       v_deq_options.dequeue_mode:=DBMS_AQ.REMOVE;
       v_deq_options.msgid:=v_msgid;
       Dbms_aq.dequeue(Queue_name=>'xxx_test_q'
                      ,Message_properties=>V_msg_properties
                      ,dequeue_options=>V_deq_options
                      ,Payload=>V_payload
                      ,Msgid=>V_msgid);
      Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
      v_deq_options.dequeue_mode:=DBMS_AQ.BROWSE;
      v_deq_options.msgid:=NULL;
   end if;
   exception
    when others then
      null;
   end;
end loop;
End;
```

This removed only the 6th message from the queue, and left the others intact.

There are more options to the queuing / dequeuing like retrying failed attempts (rollback after a dequeue is considered a failed attempt) and queuing with a delay or an expiration time. But I think the information so far will allow you to test these options on a need-by basis.

Multi-consumer or publish-subscribe queues

Both ‘publish-subscribe’ and ‘multi-consumer’ are used for these queues. I think ‘multi-consumer’ is most often used informally. That will also be the one I will use in this article (even though ‘publish-subscribe’ is more accurate).

We build multi-consumer queues with dbms_aqadm again. But on creating the queue-table, we say that it has to be a multi-consumer queue-table:

```
Begin
dbms_aqadm.create_queue_table (queue_table=>’xxx_mc_test’
                              ,multiple_consumers=>TRUE
                              ,queue_payload_type=>’RAW’);
End;
```

Now we see more objects being created. The most important ones are:

`Xxx_mc_test` The queue table itself.

Table `aq$_xxx_mc_test_s` with information about the subscribers to the queue

Table `aq$_xxx_mc_test_r` with information about the rules for the subscriptions

IOT `aq$_xxx_mc_test_h` with historic information about dequeuing
IOT `aq$_xxx_mc_test_i` with dequeuing information

As you can see, a lot more information is stored for multi-consumer queues. In part this information has to do with the subscription and subscriber mechanism. But there is also the need to keep a history of the dequeuing, to know when a message has been dequeued by all subscribers.

We will be seeing the use of all the objects in a few minutes, when we start queuing and dequeuing messages.

When we try to enqueue messages on this queue now, we receive an `ORA-24033: no recipients for message`. This means we need to set up subscribers first. If we enqueue without a recipient list, the message will be made available for all subscribers.

To add a subscriber, we use the dbms_aqadm package and a new object_type: `sys.aq$_agent`.
This type is defined as an object of name , address and protocol. The last 2 are used in inter-system communication only.

We can just call the following procedure:

```
DECLARE
V_agent sys.aq$_agent;
BEGIN
   V_agent:= sys.aq$_agent('Agent1',NULL,NULL);
   DBMS_AQADM.ADD_SUBSCRIBER(queue_name=>'xxx_mc_test_q'
                             ,subscriber=>v_agent);
END;
```

We can see the subscribers from the view `aq$xxx_mc_test_s` (or the underlying table: `aq$_xxx_mc_test_s`):

```
select * from aq$xxx_mc_test_s;

QUEUE          NAME   ADDRESS PROTOCOL TRANSFORMATION
-------------- ------ ------- -------- --------------
XXX_MC_TEST_Q  AGENT1                0
```

Now let's enqueue a message:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_enq_options Dbms_aq.enqueue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
v_payload := utl_raw.cast_to_raw('Hello world, again!');
Dbms_aq.enqueue(Queue_name=>'xxx_mc_test_q'
               ,Message_properties=>V_msg_properties
               ,enqueue_options=>V_enq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
```

```
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_msgid));
End;
```

Now when we look at the queue-view, we can see that a subscriber has been selected:

```
select queue,rawtohex(msg_id) msg_id,msg_state,consumer_name from aq$xxx_mc_test;

QUEUE          MSG_ID                           MSG_STATE CONSUMER_NAME
-------------- -------------------------------- --------- -------------
XXX_MC_TEST_Q  BC4C48AC659946428F38F8BC3AB02184 READY     AGENT1
```

Now to dequeue the message, we also need to set the consumer_name in the dequeue_options. When enqueuing a message without a subscriber_name, it can be dequeued by all subscribers. But on dequeueing, the subscriber needs to identify itself.

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
v_deq_options.consumer_name:='Agent1';
Dbms_aq.dequeue(Queue_name=>'xxx_mc_test_q'
               ,Message_properties=>V_msg_properties
               ,dequeue_options=>V_deq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
End;

Hello world, again!
```

Now when we check the queue contents, we see that the message is still there. Even after a commit, the message has been retained. Maybe you won't see it on your system immediately. But then run:

```
Begin
Dbms_aqadm.stop_time_manager;
End;
```

And enqueue/dequeue a message again. Now when you look in xxx_mc_test or aq$xxx_mc_test, you will see the message being retained (with status 'PROCESSED'). When you start the time_manager again, the message will disappear after some time.

The reason for this, is that Oracle enhances concurrency by using a separate table (IOT) for the dequeuing. When we enqueue a message again:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_enq_options Dbms_aq.enqueue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
v_payload := utl_raw.cast_to_raw('Hello world');
```

```
Dbms_aq.enqueue(Queue_name=>'xxx_mc_test_q'
               ,Message_properties=>V_msg_properties
               ,enqueue_options=>V_enq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_msgid));
End;
```

We can see the data in the dequeue-IOT:

```
select subscriber#, queue#, msg_enq_time, msgid from Aq$_xxx_mc_test_i;

SUBSCRIBER# QUEUE# MSG_ENQ_TIME                 MSG_ID
----------- ------ ---------------------------- --------------------------------
          1      0 21-03-10 14:16:00,252000000 75E41875D957455B84D80B55AE06F81C
```

Here the basic information about our message is recorded. After a subscriber dequeues the message it's version of the record is deleted only from this table (Please try this yourself, to confirm). The queue-monitors are responsible for cleaning up the queue-table after all subscribers have dequeued the message.

Now let's see what happens when we add a second subscriber for our queue:

```
DECLARE
V_agent sys.aq$_agent;
BEGIN
   V_agent:= sys.aq$_agent('Agent2',NULL,NULL);
   DBMS_AQADM.ADD_SUBSCRIBER(queue_name=>'xxx_mc_test_q'
                            ,subscriber=>v_agent);
END;
```

Any messages that were enqueued already, won't be available for this new subscriber. It can only dequeue messages enqueued after the subscriber was added.

Also you can't just change subscribers in an existing session. If you try, you will get an `ORA-25242: Cannot change subscriber name from string to string without FIRST_MESSAGE option.`

As the message describes further, you need to change the navigation of the dequeue. The default navigation is next_message, which means that Oracle will read the queue in a read-consistent and ordered way. It will take a snapshot of the queue when the first message is dequeued, and will dequeue the messages in that order. Messages that were enqueued after the first dequeue, will be read after reading all the messages in the queue. Even if priority ordering means they are enqueued earlier.

An alternative navigation is 'first_message'. When the navigation is set to 'first_message', Oracle will take a new snapshot before every dequeue, and start with the first message eligible for dequeuing. Because we change subscribers, we need to set navigation to 'First_message', to force Oracle to take a new snapshot.

(Btw. If you would try ‘first_message’ with dequeue_mode ‘Browse’, you would never get beyond the first message. Try it!)
(Btw2. The same goes for changing the filter options like correlation.)

Let’s start a new session, and enqueue a new message:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_enq_options Dbms_aq.enqueue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
v_payload := utl_raw.cast_to_raw('Hello agents!');
Dbms_aq.enqueue(Queue_name=>'xxx_mc_test_q'
               ,Message_properties=>V_msg_properties
               ,enqueue_options=>V_enq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_msgid));
End;
```

The message is still the same in xxx_mc_test:

```
select q_name, rawtohex(msgid) msg_id,state,enq_time,enq_uid from xxx_mc_test

Q_NAME        MSG_ID                           STATE ENQ_TIME                    ENQ_UID
------------- -------------------------------- ----- --------------------------- -------
XXX_MC_TEST_Q 45F11423444747B99600BCD8E9B3141E     0 21-03-10 14:33:23,783000000 SYSTEM
```

But in the queue view, we now see 2 records:

```
select queue,msg_id,msg_state,enq_time,enq_user_id,consumer_name from aq$xxx_mc_test;

QUEUE          MSG_ID                           STATE ENQ_TIME          ENQ_USER_ID CONSUMER_NAME
-------------- -------------------------------- ----- ----------------- ----------- -------------
XXX_MC_TEST_Q  45F11423444747B99600BCD8E9B3141E READY 21-03-10 14:33:24 SYSTEM      AGENT1
XXX_MC_TEST_Q  45F11423444747B99600BCD8E9B3141E READY 21-03-10 14:33:24 SYSTEM      AGENT2
```

One record for each subscriber. We can see the same in the dequeue_iot and in the history table:

```
select subscriber#,queue#,msg_enq_time,msgid from aq$_xxx_mc_test_i;

SUBSCRIBER# QUEUE# MSG_ENQ_TIME                 MSGID
----------- ------ ---------------------------- --------------------------------
          1      0 21-03-10 14:33:23,783000000 45F11423444747B99600BCD8E9B3141E
         21      0 21-03-10 14:33:23,783000000 45F11423444747B99600BCD8E9B3141E

Select Msgid,Subscriber#,Name,Dequeue_time,Dequeue_user From Aq$_xxx_mc_test_h;

MSGID                            SUBSCRIBER# NAME DEQUEUE_TIME DEQUEUE_USER
-------------------------------- ----------- ---- ------------ ------------
45F11423444747B99600BCD8E9B3141E           1    0
45F11423444747B99600BCD8E9B3141E          21    0
```

Now when we dequeue the message, the queue table is not updated:

```
Select Rawtohex(Msgid) Msg_id,State,Enq_time,Enq_uid,deq_time,deq_uid From Xxx_mc_test;

MSG_ID                           STATE ENQ_TIME                     ENQ_UID DEQ_TIME DEQ_UID
-------------------------------- ----- ---------------------------- ------- -------- -------
45F11423444747B99600BCD8E9B3141E     0 21-03-10 14:33:23,783000000 SYSTEM
```

However, the queue view reflects that the message has been dequeued by one subscriber.

```
Select Queue,Msg_id,Msg_state,Enq_time,Enq_user_id,Consumer_name From Aq$xxx_mc_test;

QUEUE         MSG_ID                           MSG_STATE ENQ_TIME          ENQ_USER CONSUMER_NAME
------------- -------------------------------- --------- ----------------- -------- -------------
XXX_MC_TEST_Q 45F11423444747B99600BCD8E9B3141E PROCESSED 21-03-10 14:33:24 SYSTEM   AGENT1
XXX_MC_TEST_Q 45F11423444747B99600BCD8E9B3141E READY     21-03-10 14:33:24 SYSTEM   AGENT2
```

The record for Agent1 has been deleted from the dequeue-IOT:

```
select subscriber#,queue#,msg_enq_time,msgid from aq$_xxx_mc_test_i;

SUBSCRIBER# QUEUE# MSG_ENQ_TIME                 MSGID
----------- ------ ---------------------------- --------------------------------
         21      0 21-03-10 14:33:23,783000000 45F11423444747B99600BCD8E9B3141E
```

And the history table also shows the dequeue:

```
Select msgid, subscriber#, Dequeue_time,Dequeue_user From Aq$_xxx_mc_test_h;

MSGID                            SUBSCRIBER# DEQUEUE_TIME                DEQUEUE_USER
45F11423444747B99600BCD8E9B3141E          21 21-03-10 14:33:23,783000000 SYSTEM
45F11423444747B99600BCD8E9B3141E           1
```

To dequeue the message for ‘Agent2’. We of course need to set the navigation to ‘First_message’:

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
V_deq_options.Wait := Dbms_aq.No_wait;
V_deq_options.Navigation:=Dbms_aq.First_message;
v_deq_options.consumer_name:='Agent2';
Dbms_aq.dequeue(Queue_name=>'xxx_mc_test_q'
               ,Message_properties=>V_msg_properties
               ,dequeue_options=>V_deq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_payload));
End;
```

Now after the QMON has processed the queue, the records will be deleted from all queues. (When a retention time has been set, the records will of course be retained for that time).

Rules for multi-consumer queues

So far we have seen different kinds of filtering for dequeuing messages. A new option comes with multi-consumer queues, where different subscribers can put a filter on their subscriptions. These filters (rules) can take the form of (complex) predicates that return a Boolean value. The rule can reference both message_properties as payload. To reference the payload, use a qualifier of 'tab.user_data'.

Let's build a new queue. To make optimal use of the 'rule'-functionality we'll use a custom type that can be referred to in the 'rules'. The type that we'll use is loosely based on the emp table.

```
create type t_emp as object
(empno   number
,ename   varchar2(10)
,job     varchar2(9)
);

BEGIN
DBMS_AQADM.CREATE_QUEUE_TABLE (queue_table => 'xxx_rule_test'
                              ,queue_payload_type => 't_emp');
DBMS_AQADM.CREATE_QUEUE(queue_name => 'xxx_rule_test_q'
                       ,queue_table => 'xxx_rule_test');
DBMS_AQADM.START_QUEUE (queue_name => 'xxx_rule_test_q');
END;
```

We add 2 subscribers to this queue.

```
DECLARE
V_agent sys.aq$_agent;
BEGIN
V_agent:= sys.aq$_agent('HR_President',NULL,NULL);
DBMS_AQADM.ADD_SUBSCRIBER(queue_name=>'xxx_rule_test_q'
                         ,subscriber=>v_agent
                         ,rule=>'tab.user_data.job=''President''');
V_agent:= sys.aq$_agent('HR_Employee',NULL,NULL);
DBMS_AQADM.ADD_SUBSCRIBER(queue_name=>'xxx_rule_test_q'
                         ,subscriber=>v_agent);
END;
```

Note how the agent 'HR_President' has a rule added to its subscription. Only messages where the job attribute of the payload is 'President' are eligible for dequeuing by this agent. Let's enqueue some messages on this queue.

```
Declare
V_payload t_emp;
V_msgid Raw(200);
V_enq_options Dbms_aq.enqueue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
v_payload := t_emp(1,'Jones','Manager');
Dbms_aq.enqueue(Queue_name=>'xxx_rule_test_q'
               ,Message_properties=>V_msg_properties
               ,enqueue_options=>V_enq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_msgid));
v_payload := t_emp(2,'King','President');
Dbms_aq.enqueue(Queue_name=>'xxx_rule_test_q'
               ,Message_properties=>V_msg_properties
```

```
               ,enqueue_options=>V_enq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(utl_raw.cast_to_varchar2(V_msgid));
End;
```

Now we have 2 messages. Only one of which matches the rule for the 'HR_President' subscriber. The 'HR_Employee' subscriber does not have any rule, and is thus eligible for all messages. We can see this when we query the queue-view:

```
select queue,rawtohex(msg_id) msg_id,msg_state,consumer_name from aq$xxx_rule_test;

QUEUE           MSG_ID                           MSG_STATE CONSUMER_NAME
--------------- -------------------------------- --------- -------------
XXX_RULE_TEST_Q 4D0FF7A800834559809AD90AFCA81444 READY     HR_EMPLOYEE
XXX_RULE_TEST_Q E5A2FDFD8EE942349E9BC9DEE88CEB10 READY     HR_EMPLOYEE
XXX_RULE_TEST_Q E5A2FDFD8EE942349E9BC9DEE88CEB10 READY     HR_PRESIDENT
```

We see that both messages are enqueued for the 'HR_Employee'. But only the message with the job 'President' is enqueued for the the 'HR_President'.

Let's dequeue the messages as 'HR_President' first, then as 'HR_Employee'.

```
Declare
V_payload t_emp;
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
v_deq_options.wait := dbms_aq.no_wait;
v_deq_options.consumer_name:='HR_President';
Dbms_aq.dequeue(Queue_name=>'xxx_rule_test_q'
               ,Message_properties=>V_msg_properties
               ,dequeue_options=>V_deq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(v_payload.empno||' '||v_payload.ename);
end;

2 King
```

This time the first message enqueued was ignored for this subscriber. Only the message that met its rule was dequeued. When dequeuing as the 'HR_Employee' both messages will be dequeued.

```
Declare
V_payload t_emp;
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
v_deq_options.wait := dbms_aq.no_wait;
v_deq_options.navigation := dbms_aq.first_message;
v_deq_options.consumer_name:='HR_Employee';
Dbms_aq.dequeue(Queue_name=>'xxx_rule_test_q'
               ,Message_properties=>V_msg_properties
               ,dequeue_options=>V_deq_options
```

```
                ,Payload=>V_payload
                ,Msgid=>V_msgid);
Dbms_output.Put_line(v_payload.empno||' '||v_payload.ename);
v_deq_options.navigation := dbms_aq.next_message;
Dbms_aq.dequeue(Queue_name=>'xxx_rule_test_q'
                ,Message_properties=>V_msg_properties
                ,dequeue_options=>V_deq_options
                ,Payload=>V_payload
                ,Msgid=>V_msgid);
Dbms_output.Put_line(v_payload.empno||' '||v_payload.ename);
end;

1 Jones
2 King
```

After these dequeues, the queue is empty for these subscribers. The only message eligible for 'HR_President' was the message with '2,King,President'. 'HR_Employee' was eligible for both messages. Remember that the 'Rule' must evaluate to a Boolean value. Valid references are to 'tab.user_data.', for object_type payloads. Also columns like 'priority' or 'correlation' from the message properties can be referenced in the rule.

Common issues with queues and troubleshooting

Above we already saw several error messages related to queues. Most of them can be expected, and should be handled in the code.

The most common issues with queues are from queues not started, or not started for enqueuing or dequeuing. The error messages for this should be quite clear, and you can just start the queue with the 'dbms_aqadm.start_queue' package. Note that when the queue is started for enqueuing or dequeuing only, you need to stop it first, then start again with the correct options enabled.

Another issue may occur because of the AQ error handling system. A dequeue with dequeue_mode 'REMOVE' that needs to roll back afterwards, is considered a failed attempt. When the number of failed attempts exceed the retry count of the queue, the message will be moved to the Exception queue. The message remains in the queue table, but with status 3: Expired. The exception_queue field will be set to the name of the exception queue.

These messages are not available for dequeuing anymore. They must be dequeued from the exception queue.

To dequeue from an exception queue, it first needs to be enabled for dequeuing. (It cannot be enabled for enqueuing). Also no subscriber_name is allowed for the dequeue.

```
Begin
Dbms_aqadm.Start_queue(Queue_name=>'aq$_xxx_mc_test_e',Enqueue=>False,Dequeue=>True);
end;
```

```
Declare
V_payload Raw(200);
V_msgid Raw(200);
V_deq_options Dbms_aq.dequeue_options_t;
v_msg_properties dbms_aq.message_properties_t;
Begin
V_deq_options.Wait := Dbms_aq.No_wait;
Dbms_aq.dequeue(Queue_name=>'AQ$_XXX_MC_TEST_E'
               ,Message_properties=>V_msg_properties
               ,dequeue_options=>V_deq_options
               ,Payload=>V_payload
               ,Msgid=>V_msgid);
Dbms_output.Put_line(Utl_raw.Cast_to_varchar2(V_payload));
End;
```

The last issue to note with queues (especially in eBS systems) is described in Metalink note 267137.1. If multi-consumer queues are created in an ASSM tablespace, or when using freelist groups, QMON will not perform space management on the IOT's. This will result in ever growing IOT's and eventually in deteriorating performance.