APPLICATION DEVELOPMENT WITH ORACLE ADVANCED QUEUING

Jeffrey Jacobs, PayPal 

INTRODUCTION

This paper and corresponding presentation are intended to provide the reader and attendee with an understanding of the basic features and capabilities of Oracle Advanced Queuing (AQ) for consideration in application development. It does not cover all of the features and capabilities of Oracle AQ. The reader is cautioned that the author is neither omniscient nor infallible. The reader should consult the Oracle documentation (***Oracle® Streams*** *Advanced Queuing User's Guide* and ***Oracle® Database*** *PL/SQL Packages and Types Reference)* prior to beginning application development.

Oracle AQ provided PL/SQL, OCI, JMS and SOAP APIs. While the all offer virtually identical functionality, this paper and presentation refers only the PL/SQL packages, DBMS_AQ and DBMS_AQADM.

WHAT IS MESSAGING?

Messaging is the ability to send a message containing data from one application/process to another application/process. It is a widely used technique in distributed systems, particularly high volume OLTP systems. Unlike client server applications, which are typically synchronous, messaging is typically asynchronous, i.e. the sender, referred to as the *producer*, is not blocked waiting for a reply from the recipient(s), referred to as *consumer(s)*. Oracle Advance Queuing (AQ) does not support synchronous messaging.

Messaging has many uses and advantages. It allows applications and systems to communicate and co-operate in an API independent manner. An order entry system may send a message containing order information to a fulfillment system without requiring access to internal APIs. The same message may also simultaneously be routed to an inventory management system, a customer support application, an email acknowledgment application, etc.

Messages are placed into queues, called *enqueuing*. The enqueuing applications are called the *producers*. There is typically no restriction on the number of producers for a given queue.

The application data portion of the message is referred to as the *payload*.

Messages are read and removed from the queue by *dequeuing* the message. Applications dequeuing messages are referred to as *consumers*.

There are three general categories of messaging:

- Single consumer, a.k.a., point-to-point - a message is dequeued by a single consumer
- Multicast - the producer effectively names designated consumers for the message
- Broadcast - consumers may dynamically gain access to a message queue by *subscribing*

A robust messaging systems supports a wide variety of features in addition to those describe above. These include:

- Error handling
- Timeouts and expirations
- Enqueuing/dequeuing of messages as a group
- Dequeueing messages by criteria other than FIFO, including, but not limited to:
  - Enqueue time
  - Priority
  - Contents of messages
- Reliability
- Propagation - pushing messages to destinations
- Other queues

- Other databases
- Other messaging systems (JMS, middleware, gateways)
- Retention of messages and history of actions
- Non-repudiation
- Logging
- Performance evaluation
- Warehousing
- Wide range of message content data types (aka *payload*), including:
  - Text
  - XML
  - BLOB, LOB, CLOB
  - Structured records
- Notification to consumers of message availability
- Guaranteed delivery
- High performance

Oracle AQ provides all of this functionality.

In addition, Oracle AQ also provides the ability to browse messages without dequeuing.

Queue Types

Oracle AQ provides the three types of messaging describe above via two basic types of queues, *single consumer queues* and *multi-consumer queues*. A multi-consumer queue may provide both multicast and broadcast capabilities.

All queues allow any application with appropriate permissions to enqueue messages.

In a single consumer queue, a given message is dequeued by only one consumer, after which it is removed from the queue. However, multiple consumers may dequeue from the queue, e.g. multiple instance of an application, such a multiple instances of a fulfillment application processing messages from a single order queue.

Single consumer queues have the simplest underlying structure and, when used appropriately, typically offer the highest performance.

Queues need to be started after creation via `START_QUEUE`. Queues can be stopped via `STOP_QUEUE`. Both procedures allow control of enqueuing and dequeuing separately.

For multi-consumer queues, the determination as to whether a message is broadcast or multicast is made at the time the message is enqueued; it is not a property of the queue itself.

Message States

A message may be in one of the following states:

- `READY` – message is available to be dequeued
- `WAITING` – availability for dequeuing is delayed
- `EXPIRED` – message has timed out and been moved to exception queue
- `PROCESSED` – message has been consumed by all consumers

Buffered Messaging

Buffered messaging is a light weight, non-persistent form of messaging, which can be specified at the time of enqueuing. It is generally only memory resident, and does not support many of the features that are available for persistent messaging. In particular, buffered messages do not support:

- Grouping
- Retention
- Guaranteed delivery

- Array dequeuing

Advanced Queuing (AQ) Tables

An AQ table is an abstract object type, which may be implemented by one or more underlying tables, indexes and index organized tables depending on whether the AQ table supports single or multi-consumer queues.

An AQ table typically holds one or more queues, which can be created and destroyed dynamically.

Multi-consumer AQ tables typically require more management and overhead.

AQ tables are created by:

```
DBMS_AQADM.CREATE_QUEUE_TABLE(
      queue_table IN VARCHAR2,
      queue_payload_type IN VARCHAR2,
      [storage_clause IN VARCHAR2 DEFAULT NULL,]
      sort_list IN VARCHAR2 DEFAULT NULL,
      multiple_consumers IN BOOLEAN DEFAULT FALSE,
      message_grouping IN BINARY_INTEGER DEFAULT NONE,
      comment IN VARCHAR2 DEFAULT NULL,
      primary_instance IN BINARY_INTEGER DEFAULT 0,
      secondary_instance IN BINARY_INTEGER DEFAULT 0,
      compatible IN VARCHAR2 DEFAULT NULL,
      secure IN BOOLEAN DEFAULT FALSE);
```

The relevant parameters are described below:

- `queue_table` – AQ table name
- `queue_payload_type` – payload type
- `storage_clause` – any valid storage clause. Tablespace should always be specified. Oracle recommends using ASSM. If ASSM is not used, INITRANS and PCTFREE may be set if needed for extremely high transaction queues; this has not been necessary in the author's experience.
- `sort_list` – determines the order in which messages are normally dequeued. It applied to all queues and governs the generation of the underlying queries. This can be overridden by certain dequeuing options, but it cannot be changed after creation. The default is enqueue time, which is effectively FIFO.
- `multiple_consumers` – `'TRUE'` or `'FALSE'`. All queues in the AQ table are of this type.
- `message_grouping` – `'NONE'` or `'TRANSACTIONAL`.

If `TRANSACTIONAL`, all messages enqueued in one transaction may be treated as a group when dequeuing. See *Transaction Protection* below.

- `comment` – a description of the AQ table which will be stored in the data dictionary.
- `primary_instance` – primary owner of the queue table service (RAC); see *RAC Considerations* below.
- `secondary_instance` – secondary owner of the queue table service (RAC); ; see *RAC Considerations* below.
- `compatible` – lowest database version compatibility (only 10gR2 and later are covered in this paper).
- `secure` – `'TRUE'` for secure queues (not covered in this paper).

RAC Considerations

Each AQ table effectively creates a service. AQ table structures are typically hot tables with a great potential for hot blocks. To avoid performance issues caused by cache contention, the services should be pinned to a single node (aka *node affinity).*

`primary_instance` specifies the preferred instance on which the service will run. `secondary_instance` - specifies the preferred instance if primary instance is not available. If neither instance is available, a "random" instance is selected.

Creating Queues

Queues are created via:

```
DBMS_AQADM.CREATE_QUEUE (
      queue_name IN VARCHAR2,
      queue_table IN VARCHAR2,
```

```
      queue_type IN BINARY_INTEGER DEFAULT NORMAL_QUEUE,
      max_retries IN NUMBER DEFAULT NULL,
      retry_delay IN NUMBER DEFAULT 0,
      retention_time IN NUMBER DEFAULT 0,
      dependency_tracking IN BOOLEAN DEFAULT FALSE,
      comment IN VARCHAR2 DEFAULT NULL,
      auto_commit IN BOOLEAN DEFAULT TRUE);
```

The parameters are described below:

- `queue_name` – the name of the queue.
- `queue_table` – the name of the AQ table holding queue.
- `queue_type` – `NORMAL_QUEUE` or `EXCEPTION_QUEUE`.
- `max_retries` – the maximum number of dequeue retries before moving to exception queue; see *Transaction Protection* below.
- `retry_delay` – after a failure (usually `ROLLBACK`), the number of seconds before message will be available for dequeuing again.
- `retention_time` – the time the message remains in the *queue table* after dequeuing.
- `dependency_tracking` - not currently implemented
- `comment` – Queue documentation, which is kept in the data dictionary.
- `auto_commit` - deprecated;

Enqueue Options and Features

There is a wide range of options for enqueuing messages. These options include, but are not limited to:

- Enqueuing a single message.
- Enqueuing an array of messages (PL/SQL or OCI).
- Message Grouping, which treats all messages enqueued in a single transaction as a group.
- Sender Identification.
- Time Specification and Scheduling of message delivery.
- Correlation Identifier, which allows multiple messages queued with a user defined identifier to be dequeued together.

Enqueuing Message

The following PL/SQL API is used to enqueue messages:

```
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);
```

- `queue_name` – the name of the queue in which the message is to be enqueued.
- `payload` - the type definition of the payload, typically, but not limited to, a PL/SQL abstract type
- `msg_id` - the unique identifier of the message

DBMS_AQ.ENQUEUE_OPTIONS_T

The `DBMS_AQ.ENQUEUE_OPTIONS_T` record contains the options for enqueuing the message as described below:

```
TYPE SYS.ENQUEUE_OPTIONS_T IS RECORD (
      visibility BINARY_INTEGER DEFAULT ON_COMMIT,
      relative_msgid RAW(16) DEFAULT NULL,
      sequence_deviation BINARY_INTEGER DEFAULT NULL,
      transformation VARCHAR2(61) DEFAULT NULL,
      delivery_mode PLS_INTEGER NOT NULL DEFAULT PERSISTENT);
```

The attributes are:

- visibility
  - ON_COMMIT - the message is enqueued as part of the transaction, i.e. enqueuing the message is completed by COMMIT.
  - IMMEDIATE – the message is enqueued immediately in an autonomous transaction.
- transformation - Specifies a transformation function to be performed before enqueuing (not covered in this paper).
- delivery_mode
  - PERSISTENT - the message is stored in the queue table.
  - BUFFERED - the message is only maintained in memory, and may be lost in the event of system failure or database shutdown.
- sequence_deviation - deprecated as of 10.2
- relative_msg_id – effectively deprecated.
- sequence_deviation – effectively deprecated.

DBMS_AQ.MESSAGE_PROPERTIES_T

The DBMS_AQ.MESSAGE_PROPERTIES_T record is used for both enqueuing and dequeuing operations

```
TYPE message_properties_t IS RECORD (
      priority BINARY_INTEGER NOT NULL DEFAULT 1,
      delay BINARY_INTEGER NOT NULL DEFAULT NO_DELAY,
      expiration BINARY_INTEGER NOT NULL DEFAULT NEVER,
      correlation VARCHAR2(128) DEFAULT NULL,
      attempts BINARY_INTEGER,
      recipient_list AQ$_RECIPIENT_LIST_T,
      exception_queue VARCHAR2(61) DEFAULT NULL,
      enqueue_time DATE,
      state BINARY_INTEGER,
      sender_id SYS.AQ$_AGENT DEFAULT NULL,
      original_msgid RAW(16) DEFAULT NULL,
      signature aq$_sig_prop DEFAULT NULL,
      transaction_group VARCHAR2(30) DEFAULT NULL,
      user_property SYS.ANYDATA DEFAULT NULL
      delivery_mode PLS_INTEGER NOT NULL DEFAULT DBMS_AQ.PERSISTENT);
```

The relevant enqueue attributes are:

- priority – the priority of the message. This is only relevant if the sorting method specified for the table includes the priority.
- delay – specifies number of seconds before a message is available for dequeuing. Default is 0 (NO_DELAY)
- expiration – the number of seconds a message is available for dequeuing (after delay). If the message is not dequeued by all subscribers, it will be moved to the exception queue with a status of EXPIRED. This is necessary for multi-consumer queues, as not all subscribers may be able to dequeue the message. Default is the constant NEVER.

  delivery_mode - DBMS_AQ.BUFFERED or DBMS_AQ.PERSISTENT, determines if the message is buffered or persistent. The default is persistent.
- correlation - the ID used for dequeuing by correlation ID. This is a producer supplied value, which allows a logical grouping of messages. Unlike a transaction group, the messages need not be enqueued in a single transaction or by the same producer.

DEQUEUING FEATURES

Oracle AQ provides very high performance and functionality. Key features include:

- Concurrent dequeues

- Multiple dequeue methods and options
- Array dequeue
- Message navigation
- Waiting for messages
- Retries with delays
- Transaction protection
- Exception queues

DBMS_AQ.DEQUEUE

The PL/SQL API is:

```
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);
```

Note that `message_properties_t` is used for both enqueue and dequeue operations.

DEQUEUE_OPTIONS_T

```
TYPE DEQUEUE_OPTIONS_T IS RECORD (
      consumer_name VARCHAR2(30) DEFAULT NULL,
      dequeue_mode BINARY_INTEGER DEFAULT REMOVE,
      navigation BINARY_INTEGER DEFAULT NEXT_MESSAGE,
      visibility BINARY_INTEGER DEFAULT ON_COMMIT,
      wait BINARY_INTEGER DEFAULT FOREVER,
      msgid RAW(16) DEFAULT NULL,
      correlation VARCHAR2(128) DEFAULT NULL,
      deq_condition VARCHAR2(4000) DEFAULT NULL,
      signature aq$_sig_prop DEFAULT NULL,
      transformation VARCHAR2(61) DEFAULT NULL,
      delivery_mode PLS_INTEGER DEFAULT PERSISTENT);
```

The `DBMS.AQ.DEQUEUE_OPTIONS_T` specifies the dequeuing options as described below:

- `consumer_name` – the name of the subscriber.
- `dequeue_mode. Modes include:`
  - `REMOVE` (with data) – this is the typical dequeuing method. The message may remain in the queue table for history based on retention period, but it not eligible for future dequeuing (unless via `msg_id`).
  - `REMOVE_NODATA` – no data is returned, but the message is removed from queue. This may be used for selective cleanup.
  - `BROWSE` – reads the message data, but does not actually dequeue the message. The message remains available for future processing (unless dequeued by another process). Browsing may not be repeatable, and as such there are numerous "gotchas" to be aware of.
- `navigation` – there are two methods for ***`navigation`*** when dequeuing.
  - `FIRST_MESSAGE` - This creates a "snapshot" (effectively a cursor); note that this only retrieves messages that were enqueued at the time of the dequeue call.
  - `NEXT_MESSAGE` – If `FIRST_MESSAGE` was used, this retrieves the next message in the snapshot. See *Default Dequeuing* below.
- `wait` – if no messages are available, the consumer may wait for the next message. The options are:
  - `FOREVER` – waits forever, which is the default. Typically used for high frequency queues. Note that this blocks the process.

- NO_WAIT – don't wait for next message. Typically used for deferred or batch operations, which are initiated by jobs scheduled at regular intervals.
- Number – the wait time in seconds. Process is blocked while waiting.

The next message is dequeued on wake up.

NOTE BENE: Oracle AQ also offers the ability for a process to *listen* on multiple queues; the functionality is outside the scope of this paper.

DEQUEUE METHODS

There are several methods for dequeuing messages. The default is to dequeue individual messages based on the sort order specified when the AQ table was created.

NOTE BENE: the most efficient navigation method for dequeuing based on the sort order is to use NEXT_MESSAGE without FIRST_MESSAGE. FIRST_MESSAGE *always* performs a query. However, if NEXT_MESSAGE is used *without* FIRST_MESSAGE, it will only perform one SELECT in the session; subsequent calls are simple fetches.

Other methods are:

- Correlation ID – dequeue series of message based on correlation as follows:
  - Get correlation id by dequeuing using FIRST_MESSAGE. Dequeue additional messages via NEXT_MESSAGE using the value of correlation until no more messages remain.
    The specification for correlation may use pattern matching (%,_).

This method typically requires the addition of an index and generation of statistics to force the underlying queries to use the index on the correlation column..

- Transaction group – similar to correlation, but uses transaction_group set by producer. Should use array dequeuing, but may use same loop as Correlation ID above, but specifying the transaction_group. Pattern matching may also be used.
- deq_condition– similar to SQL WHERE clause, accesses contents of payload object elements or other columns. See documentation for more details about specifying columns and payload elements. Note that using the method supersedes all other methods.
- msgid - dequeue a single message by system-assigned RAW value. This typically requires browsing the queue(s), and is usually used for cleanup and corrections.

DEQUEUE VISIBILITY

Messages may be dequeued in the following modes:

- IMMEDIATE – Messages are removed from the queue in an autonomous transaction. If the application does not have retry capabilities, this will typically offer better performance and scalability
- ON_COMMIT (transaction protection) - Messages are removed from the queue on COMMIT of the transaction. The dequeue operation is treated in the same manner as an INSERT/UPDATE/DELETE. If the transaction fails, either due to ROLLBACK, system failure or shutdown, the retry count is incremented. If the retry count is exceeded, the message is moved to the exception queue, otherwise it remains in the original queue. Note that a system failure or shutdown may not increment the retry count. If retry_delay was specified when the queue was created, the message will not be available for dequeuing for the specified number of seconds.

MESSAGE EXPIRATION

If expiration is specified in message_properties_t.expiration, all consumers must dequeue the message before expiration time. Otherwise, the message is moved to the exception queue. It is generally a good practice to specify expiration for multi-consumer queues, as not all consumers may be active, which would result in the message remaining in the queue indefinitely.

Exception Queues

Each AQ table has at least one exception queue which contains messages that have expired or exceeded retry count from all of the other queues. Messages in an exception queue may be dequeued *once* by only one consumer for reprocessing. Exception queues should be monitored and periodically emptied either for reprocessing or simply free space.

Propagation

Messages may be *pushed* to other queues via *propagation*. Those queues typically, but not always, exist in another database or an external messaging system; the latter is beyond the scope of this paper. Propagation may also be to queues in the same database. The messages are ultimately processed by consumers of the destination queue(s); propagated messages are considered process upon completion of propagation. Propagation may push messages to multiple queues in multiple targets (fan out). Messages may also be propagated from multiple sources into a single queue. The destination queue may be single or multi-consumer, but must be of the same payload type. Propagation is performed by scheduled jobs. A *propagation window* is a period of time in which propagation can occur, i.e. effectively scheduling the job.

There are two basic modes for propagation between databases:

- Queue to dblink – Effectively deprecated.
- Queue to queue – the target queues are specified.

The API to schedule propagation is:

```
DBMS_AQADM.SCHEDULE_PROPAGATION (
      queue_name IN VARCHAR2,
      destination IN VARCHAR2 DEFAULT NULL,
      start_time IN DATE DEFAULT SYSDATE,
      duration IN NUMBER DEFAULT NULL,
      next_time IN VARCHAR2 DEFAULT NULL,
      latency IN NUMBER DEFAULT 60,
      destination_queue IN VARCHAR2 DEFAULT NULL);
```

The parameters are:

- `queue_name` – the name of the queue to be propagated.
- `destination` – destination dblinks.
- `start_time` – the start time for the propagation, i.e. the time when the job will first be schedule.
- `duration` – how long propagation lasts in seconds. NULL means the propagation lasts forever (or until stopped or altered).
- `next_time` – a calendar expression (as used by DBMS_SCHEDULER) for the next propagation window.
- `latency` – if no messages, how many seconds to wait until checking the queue for message to be propagated. 0 results in propagation as soon as a message is available.

Other APIs to manage propagation are:

- `ALTER_PROPAGATION_SCHEDULE`
- `DISABLE_PROPAGATION_SCHEDULE`
- `ENABLE_PROPAGATION_SCHEDULE`
- `SCHEDULE_PROPAGATION`
- `VERIFY_QUEUE_TYPES`

AQ Table Structures

A multi-consumer AQ table has 7 underlying tables, both heap and index organized. The main table with message data for all queues has the same name as specified in `CREATE_QUEUE_TABLE`, e.g. `ORDERS_QUEUETABLE`. Other tables have names beginning with AQ$, e.g. `AQ$_ORDERS_QUEUETABLE_H`

A single consumer AQ table creates a single table with main table name; the index structure may vary.

Performance Tips for Dequeuing

Using certain features, such as correlation id or transaction grouping, may require additional indexes on the main table. To change the behavior of the queries used by AQ, statistics need to be gathered, as AQ tables are exempt from automatic statistics gathering. However, generating appropriate statistics in a production environment can be problematic due to the volatile nature of queues; stopping the queues to allow messages to build up in order to gather statistics is probably not acceptable to the DBAs. Statistics can either be created manually, or, in a development or QA environment, messages can be enqueued without dequeuing. The statistics can then be imported into production for the table. It's also a good idea to lock the statistics, just to be safe.

Query to be Tuned

Finding the underlying dequeuing query for tuning is not immediately obvious. Look in appropriate V$ or GV$ views or AWR report for the following pattern:

```
SELECT     /*+ FIRST_ROWS(1) */
        tab.ROWID,
        …
         tab.user_data
    FROM <queue_table_name> -- the name of the main queue table
   WHERE q_name = :1 AND (state = :2 and …
ORDER BY q_name, …
FOR UPDATE SKIP LOCKED;
```

FOR UPDATED SKIP LOCKED is the "secret sauce" for AQ's performance. It performs SELECT FOR UPDATE *only on rows that are not currently locked!!!* It also apparently only locks rows when they are fetched, but this has been difficult to confirm. This is not a documented or user supported feature.

More Stuff

It is not possible to cover all of the capabilities and functionality of Oracle AQ in this paper. Some other features of potential interested include:

- AQ automatically manages space, perform `COALESCE` as well as removing messages that have passed their retention periods.
- There are numerous APIs for managing all aspects of AQ.
- AQ can propagate messages via external protocols and gateways
- AQ can be accessed via SOAP
- AQ can retain the entire history of a message for non-repudiation, logging, etc.

The author strongly urges the reader to consult the appropriate documentation, in particular ***Oracle® Database*** *PL/SQL Packages and Types Reference* and ***Oracle® Streams*** *Advanced Queuing User's Guide.*