The On-Commit Database Trigger

Philipp Salvisberg
Trivadis AG
Zürich

Keywords

Oracle Database Development, PL/SQL, SQL, Java, JMS, Advanced Queuing (AQ), Spring Boot, Twitter4J

Introduction

RESTful Web Services are a convenient way to access services of any kind. You may call such services from PL/SQL, but you lose all the nice transaction processing features of the database. Oracle Advanced Queuing (AQ) may participate in a database transaction and is well-suited to integrate any service into a PL/SQL based application. Basically, AQ makes an on-commit behavior possible.

This paper contains simplified code excerpts. The complete source code is available in my GitHub repository on https://github.com/PhilippSalvisberg/emptracker.

Use Case

The most famous table in the Oracle world is the table EMP. The 14 rows are well known and stable since the early 80's.

A salary change is something a lot of people would be interested in, right? So, let's track salary changes in table EMP and tweet about it. But we want to communicate only committed changes. In fact, the following transaction should not cause any action:

```
UPDATE emp SET sal = sal * 2;
ROLLBACK;
```

And the next transaction should lead to 5 tweets even if 6 rows are updated.

```
UPDATE emp SET sal = sal + 100 WHERE JOB = 'SALESMAN';
UPDATE emp SET sal = sal + 200 WHERE ename IN ('MARTIN', 'SCOTT');
COMMIT;
```

```
ENAME      JOB              SAL    NEW_SAL
---------- --------- ---------- ----------
SMITH      CLERK            800        800
ALLEN      SALESMAN        1600       1700 +100
WARD       SALESMAN        1250       1350 +100
JONES      MANAGER         2975       2975
MARTIN     SALESMAN        1250       1550 +300
```

```
ENAME      JOB              SAL     NEW_SAL
---------- --------- ---------- ----------
BLAKE      MANAGER          2850       2850
CLARK      MANAGER          2450       2450
SCOTT      ANALYST          3000       3200 +200
KING       PRESIDENT        5000       5000
TURNER     SALESMAN         1500       1600 +100
ADAMS      CLERK            1100       1100
JAMES      CLERK             950        950
FORD       ANALYST          3000       3000
MILLER     CLERK            1300       1300
```

Please note that MARTIN got an aggregated salary increase of $300.

Solution Architecture

In the first transaction of the proposed solution, we capture all 6 updates in a database trigger and produce 6 messages in the RAW queue. In the second transaction, we consume all messages of the first transaction, aggregate the salary changes for every employee and store 5 aggregated messages in the AGGR queue. Afterwards a Java application consumes the aggregated messages from the AGGR queue. Every message is consumed in a single transaction. Posting a tweet is part of a transaction. The following picture visualizes the solution architecture:

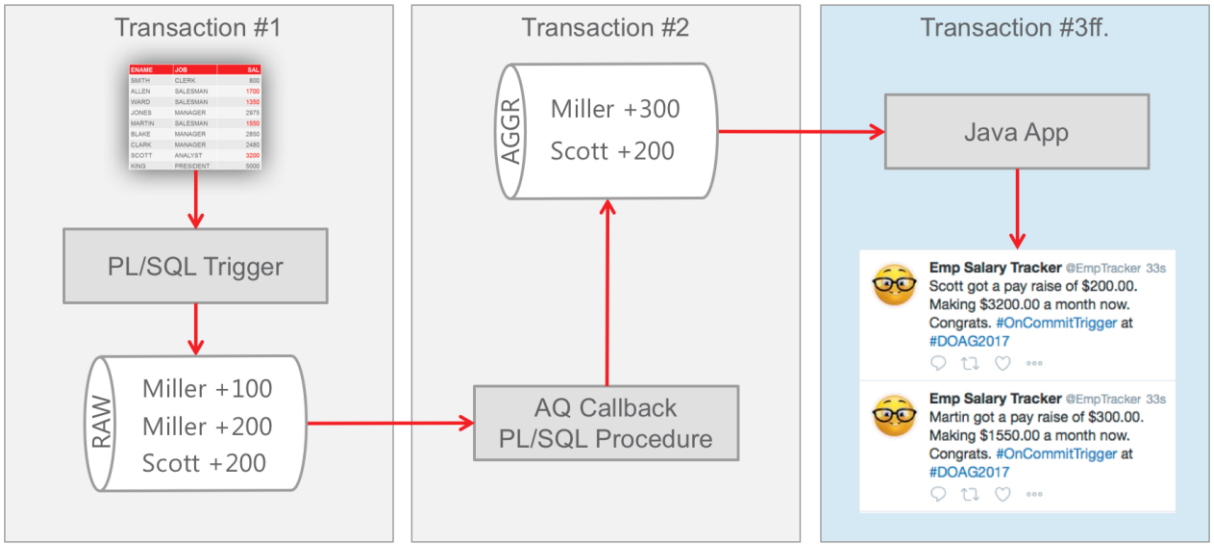

Illustration 1: Solution Architecture

Transaction #1 and #2 are implemented using Oracle Database features. We use SQL to aggregate the messages from the RAW queue. The performance impact on transaction #1 is minimized. Transaction #1 and #2 are not relying on an Internet connection nor on the availability of Twitter. This makes the solution robust. We use a well-established Java library for the Twitter API. There is no added value to rewrite this in PL/SQL or load the Java library with all its dependencies into the database.

Database Setup for PL/SQL and JMS

We create the queue tables based with the payload type SYS.AQ$_JMS_TEXT_MESSAGE. This simplifies the use of AQ from Java a lot, because we may rely on the Java Message Service (JMS) and all its tooling.

```
BEGIN
   dbms_aqadm.create_queue_table (
      queue_table        => 'REQUESTS_QT',
      queue_payload_type => 'SYS.AQ$_JMS_TEXT_MESSAGE',
      sort_list          => 'PRIORITY,ENQ_TIME',
      multiple_consumers => TRUE,
      message_grouping   => dbms_aqadm.transactional
   );
   dbms_aqadm.create_queue_table (
      queue_table        => 'RESPONSES_QT',
      queue_payload_type => 'SYS.AQ$_JMS_TEXT_MESSAGE',
      sort_list          => 'PRIORITY,ENQ_TIME',
      multiple_consumers => TRUE
   );
END;
/
```

The request queue table uses transactional message grouping. This means that all messages get a transaction identifiers and a transaction step number. This information is used to dequeue messages of a transaction in the original order. The queues are created with a retention time of a week. This way the messages will be deleted from the queue after a week and we get a managed log for free. The longer retention time is very helpful for debug purposes. We configured also a retry after 2 seconds. In real live scenarios, you would configure a higher retry delay to give the failing system enough time to recover.

```
BEGIN
   dbms_aqadm.create_queue (
      queue_name          => 'REQUESTS_AQ',
      queue_table         => 'REQUESTS_QT',
      max_retries         => 1,
      retry_delay         => 2, -- seconds
      retention_time      => 60*60*24*7 -- 1 week
   );
   dbms_aqadm.create_queue (
      queue_name          => 'RESPONSES_AQ',
      queue_table         => 'RESPONSES_QT',
      max_retries         => 1,
      retry_delay         => 2, -- seconds
      retention_time      => 60*60*24*7 -- 1 week
   );
END;
/
```

Finally, we need to enable the queues for enqueue and dequeue operations.

```
BEGIN
   dbms_aqadm.start_queue(
      queue_name => 'REQUESTS_AQ'
     ,enqueue    => TRUE
     ,dequeue    => TRUE
   );
   dbms_aqadm.start_queue(
      queue_name => 'RESPONES_AQ'
     ,enqueue    => TRUE
     ,dequeue    => TRUE
   );
END;
/
```

PL/SQL Application Using JMS

In a publish-subscribe scenario we must register subscribers for topics of interest. It is not possible to enqueue a message without a subscriber. The queuing system needs to know when a message is consumed by all subscribers and may be deleted after the defined retention period.

In the PL/SQL application we just read messages from the RAW queue. But how are we going to distinguish between RAW and AGGR messages when we have set up a single queue for all messages? We use a rule, which is basically a filter on the queue table. The alias tab is referring to the queue table and the column user_data contains the payload, an instance of SYS.AQ$_JMS_TEXT_MESSAGE.

```
BEGIN
   -- for messages created in trx #1 to be processed in trx #2
   dbms_aqadm.add_subscriber(
      queue_name => 'requests_aq',
      subscriber => sys.aq$_agent('RAW_ENQ', 'REQUESTS_AQ', 0),
      rule       => q'[tab.user_data
                       .get_string_property('msg_type') = 'RAW']'
   );
   -- for response/logging messages produced in trx #2 and #3ff.
   dbms_aqadm.add_subscriber(
      queue_name => 'responses_aq',
      subscriber => sys.aq$_agent('ALL_RESPONSES','RESPONSES_AQ', 0)
   );
END;
/
```

We use the response queue mostly for logging purposes. We do not plan to dequeue messages from the response queue. But nonetheless it is necessary to register a subscriber, in this case without a filter.

Now we create the database trigger to enqueue messages as part of transaction #1. Please note that we add a property named msg_type with the value RAW to each message. This is necessary to match the filter criterion for the subscriber RAW_ENQ. But we still handle the exception when no subscribers are defined for a message. The idea is to make the application robust. Updating the salary should be possible even if there is no interest in notification.

```
CREATE OR REPLACE TRIGGER emp_au_trg
   AFTER UPDATE OF sal ON emp
   FOR EACH ROW
DECLARE
   PROCEDURE enqueue IS
      l_enqueue_options sys.dbms_aq.enqueue_options_t;
      l_message_props   sys.dbms_aq.message_properties_t;
      l_jms_message     sys.aq$_jms_text_message :=
                           sys.aq$_jms_text_message.construct;
      l_msgid           RAW(16);
      e_no_recipients   EXCEPTION;
      PRAGMA exception_init(e_no_recipients, -24033);
   BEGIN
      l_jms_message.clear_properties();
      l_message_props.correlation := sys_guid;
      l_message_props.priority := 3;
      l_message_props.expiration := 30; -- 30 seconds
      l_jms_message.set_replyto(
         sys.aq$_agent('ALL_RESPONSES', 'RESPONSES_AQ', 0));
      l_jms_message.set_string_property('msg_type', 'RAW');
      l_jms_message.set_string_property('ename', :old.ename);
      l_jms_message.set_double_property('old_sal', :old.sal);
      l_jms_message.set_double_property('new_sal', :new.sal);
      sys.dbms_aq.enqueue(
         queue_name         => 'requests_aq',
         enqueue_options    => l_enqueue_options,
         message_properties => l_message_props,
         payload            => l_jms_message,
         msgid              => l_msgid
      );
   EXCEPTION
      WHEN e_no_recipients THEN
         NULL; -- OK, topic is not of interest
   END enqueue;
BEGIN
   IF :old.sal != :new.sal THEN
      enqueue;
   END IF;
END;
/
```

In the next step, we register a callback procedure, to process every message produced in transaction #1 automatically. This is like a database trigger which fires on-commit for each row. This way we do not need to define a background job to listen for new messages. Oracle does that for us. It is called event-based notification and is managed by the EMON coordinator background process (EMNC).

```
BEGIN
   dbms_aq.register(
      reg_list => sys.aq$_reg_info_list(
                     sys.aq$_reg_info(
                       name      => 'REQUESTS_AQ:RAW_ENQ',
                       namespace => dbms_aq.namespace_aq,
                       callback  => 'plsql://raw_enq_callback?PR=0',
                       context   => NULL
                     )
                  ),
      reg_count => 1
   );
END;
/
```

The callback procedure raw_enq_callback is called for each message, but the first call of dequeue_all will consume all messages within a group to do the aggregation using SQL in the procedure enqueue_aggr_messages.

```
CREATE OR REPLACE PROCEDURE raw_enq_callback (
   context  IN RAW,
   reginfo  IN SYS.AQ$_REG_INFO,
   descr    IN SYS.AQ$_DESCRIPTOR, -- queue, consumer, msg_id, …
   payload  IN RAW,
   payloadl IN NUMBER
) IS
   …
BEGIN
   dequeue_all;
   copy_id_to_jms_messages;
   enqueue_aggr_messages;
EXCEPTION
   …
END raq_enq_callback;
/
```

Here's the excerpt of the dequeue_all procedure.

```
l_dequeue_options.consumer_name := descr.consumer_name;
l_dequeue_options.navigation := sys.dbms_aq.first_message_one_group;
l_dequeue_options.wait       := sys.dbms_aq.no_wait;
l_msg_count := dbms_aq.dequeue_array(
   queue_name               => descr.queue_name,
   dequeue_options          => l_dequeue_options,
   array_size               => 1000,
   message_properties_array => t_message_props,
   payload_array            => t_jms_message,
   msgid_array              => t_msgid
);
```

We dequeue a maximum of 1000 messages in one call. The result is stored in the payload array t_jms_message. This is a TABLE OF SYS.AQ$_JMS_TEXT_MESSAGE and we query this collection type with SQL in enqueuer_aggr_messages. Here's the code excerpt:

```
FOR r IN (
   WITH
      msg AS (
         SELECT m.get_int_property('id') AS id,
                m.get_string_property('ename') AS ename,
                m.get_double_property('old_sal') AS old_sal,
                m.get_double_property('new_sal') AS new_sal
           FROM TABLE(t_jms_message) m
      ),
      base AS (
         SELECT MAX(id) OVER(PARTITION BY ename) AS id,
                ename,
                FIRST_VALUE(old_sal) OVER(
                   PARTITION BY ename ORDER BY id) AS old_sal,
                LAST_VALUE(new_sal) OVER(
                   PARTITION BY ename ORDER BY id
                   ROWS BETWEEN UNBOUNDED PRECEDING
                            AND UNBOUNDED FOLLOWING) AS new_sal
           FROM msg
      )
      SELECT DISTINCT id, ename, old_sal, new_sal
        FROM base
       WHERE old_sal != new_sal
) LOOP …
```

There are various ways to do the aggregation, but one thing should become clear. SQL is very well suited to do this job.

Java Application Using JMS

We use Spring Boot to write the application consuming all aggregated queue messages. The main class EmptrackerApplication looks quite simple:

```
@SpringBootApplication
public class EmptrackerApplication {
   public static void main(String[] args) {
      SpringApplication.run(EmptrackerApplication.class, args);
   }
}
```

Spring will find the AppConfig class and the application.properties file. The configured bean msgListenerContainer listens for messages with the msg_type AGGR or TWEET. Every consumed message is passed to the configured instance of the TextMessageListener class.

```
@Configuration
public class AppConfig {
   …
   @Bean
   public DefaultMessageListenerContainer msgListenerContainer() {
      DefaultMessageListenerContainer cont =
         new DefaultMessageListenerContainer();
      cont.setMessageListener(new TextMessageListener());
      cont.setConnectionFactory(
       AQjmsFactory.getTopicConnectionFactory(messageDataSource()));
      cont.setDestinationName("requests_aq");
      cont.setPubSubDomain(true);
      cont.setSubscriptionName("EmpTracker");
      cont.setSubscriptionDurable(true);
      cont.setMessageSelector("msg_type IN ('AGGR', 'TWEET')");
      cont.setSessionAcknowledgeMode(Session.SESSION_TRANSACTED);
      cont.setSessionTransacted(true);
      cont.setConcurrency("1-4");
      cont.setMaxMessagesPerTask(20);
      cont.setReceiveTimeout(10);
      return cont;
   } …
}
```

The TextMessageListener class implements the onMessage method which gets the JMS TextMessage, the Java counterpart of the SYS.AQ$_JMS_TEXT_MESSAGE object type.

The AQ message is locked until the session is committed or rollbacked. In this example, you see that the commit happens after a sending the response message.

```
@Component
public class TextMessageListener
   implements SessionAwareMessageListener<TextMessage> {…
   public void onMessage(final TextMessage request,
      final Session session) {
      try {
         String text = request.getText();
         if (text == null || text.isEmpty()) {
            String ename = request.getStringProperty("ename");
            Double oldSal = request.getDoubleProperty("old_sal");
            Double newSal = request.getDoubleProperty("new_sal");
            text = getText(ename, oldSal, newSal);
         }
         Status status = twitter.updateStatus(text);
         String screenName = status.getUser().getScreenName();
         sendResponse(request, session,
            screenName + ": " + text, "INFO");
         session.commit();
      } catch (Exception e) {…}
   }
}
```

Core Messages

Consider carefully what belongs into an Oracle Database. Just because you can do it in the database does not necessarily mean it is the best to do it there.

Oracle AQ is a cost-free option and part of every database edition. It is very well suited to emulate an on-commit trigger behavior. Just use it.

Contact address:

Philipp Salvisberg
Trivadis AG
Sägereistrasse 29
CH-8152 Glattbrugg (Zürich)

Phone:	+41-58-459 55 55
Fax:	+41-58-459 55 95
Email	philipp.salvisberg@trivadis.com
Web:	https://www.trivadis.com/
Blog:	https://www.salvis.com/blog/