Summary Table Management By Advanced Queues

By Arup Nanda, Proligence Inc.

Abstract

One of the biggest challenges a designer in a data warehouse faces demanding near real time synchronization is maintaining summary tables. While it becomes imperative to incrementally refresh the changes from the source tables, the traditional methods like materialized views fail when the summary building query becomes extremely complex in nature with several joins, especially outer joins. The complete refresh may not be acceptable due to the real time performance needs. Using triggers to populate the summary table does not work either, due to the data concurrency requirements. This paper presents a rather unconventional approach to resolve this issue using *Oracle Advanced Queues*. The solution is to feed the changes to the source tables from multiple processors to a queue and at the other end, have only one process retrieving the information and applying the changes to the summary tables. This approach takes care of a concurrency problem while attaining a near real time performance.

Background

In this actual case, we had a very complex query the client applications issued before processing other records. Here is the query. Please note, the tables and columns have been changed to protect the intellectual property.

```
SELECT DISTINCT NVL(C.COL1, R.COL1) AS COL1,
RQ.COL2, COUNT(C.COL3) AS COUNTER
FROM TAB1 R, TAB2 C, TAB3 CP, TAB4 RQ
WHERE R.COL4 = C.COL4
AND C.COL1 = RQ.COL1
AND   ( CP.COL5 = :b0
OR (CP.COL5 = :b1 AND CP.COL6 = :b2))
AND CP.COL3 = C.COL3
AND NOT EXISTS
(SELECT 1
FROM TAB5 CCL,TAB6 CL, TAB7 CHO
WHERE CCL.COL7 = CL.COL7
AND CL.COL8 = CHO.COL8
AND CCL.COL3 = C.COL3
AND CHO.COL1 = 'Y')
GROUP BY NVL(C.COL1, R.COL1), RQ.COL2
ORDER BY RQ.COL2
```

The initial design was by creating a view from the query and letting the users select from the view. This proved costly in terms performance. Running the query in sqlplus took about 40 minutes to run, and taking up huge temporary tablespace areas. Since this query was supposed to be run very frequently, this approach could not be used.

The second approach was to use a materialized view refresh on commit for the query. Since the query was complex, it could not be set up for fast refresh. A complete refresh was taking more than 24 hours and thus it was not feasible. Other options considered along this line were building materialized views from materialized views and making each one fast refreshable on commit. However, the maintenance of several layers of materialized views would have been expensive in terms of personnel resources and space. A bigger problem in on commit fast refreshable MVs was the result of a failure. In this case, if a refresh ever failed, the future fast refreshes will be stopped until the DBA made a call to DBMS_MVIEW package. Being of high transaction rate nature, this type of refresh was very much prone to errors and the overall solution would have proved too expensive to maintain. In addition, when a prototype was built and tested at the expected transaction rate, the ultimately materialized view could not be refreshed within acceptable time limit.

Ultimately the option that appeared somewhat acceptable was to create triggers on all source tables to capture changes and update the summary table built earlier. Since the query's nature was known, it was easy to assemble the code that updated the counts in the summary table when a record from source tables is changed in such way that changed its selectivity in the query. Let's see how this approach was different.

In order to present the concepts of this approach, it would be overwhelming to present the actual solution for the case given above. Therefore, it will be illustrated using a much-simplified query based on the sample tables from the SCOTT schema. Here is the query.

```
SELECT DEPTNO, COUNT(*)
FROM EMP
WHERE STATUS = 'ACTIVE'
GROUP BY DEPTNO
```

Of course, this is not a complicated query at all; but this will suffice for the demonstration. Since this query was taking a long time to execute, we built a table called DEPT_COUNTS with two columns, DEPTNO and EMP_COUNT. The table was initially populated by a query. A trigger was created on the EMP table that updated the EMP_COUNT when the rows were changed, deleted or inserted. The pseudo code was like the following.

```
If INSERTING then
   If STATUS = 'ACTIVE' then
      If (record present in DEPT_COUNTS for this deptno) then
      Update DEPT_COUNTS
      Set EMP_COUNTS = EMP_COUNTS + 1;
   Else
      Insert into DEPT_COUNTS values
      (this Deptno, 1);
      end if
   end if;
```

...

The code was then extended for deletions and updates in the same manner. Since this trigger manipulated the counts, the needed data therefore could be obtained from the DEPT_COUNTS table without using the complex query.

This approach was fine in principle and would have worked *but in a single user environment only*. Let's examine this method closely. If two sessions insert employee records for the same department, for which there were no records before, then we will have the following situation.

Time	Action
0	There is no employee record with DEPTNO = 5; therefore there is no record in DEPT_COUNTS for DEPTNO = 5
1	Session 1 inserts record with DEPTNO = 5; does not commit.
2	The trigger finds no record for DEPTNO = 5 in the DEPT_COUNTS table. This inserts a record for DEPTNO = 5 in DEPT_COUNTS table.
3	Session 2 inserts another record with DEPTNO = 5
4	The trigger sees the read consistent view of the table DEPT_COUNTS. In this read consistent view, it also finds no record for DEPTNO = 5 in the DEPT_COUNTS table. This inserts a record for DEPTNO = 5 in DEPT_COUNTS table. However, since the primary key is DEPTNO, and for DEPTNO = 5 there is another row, albeit uncommitted, this transaction waits for a lock.
5	Session 1 Commits. The committed value of EMP_COUNTS = 1.
6	Session 2 errors out, since it also tried to insert the same record, DEPTNO = 5 and EMP_COUNTS = 1.

As you can see the second transaction fails where it should not have. A similar problem occurs when a session deletes the last employee of a department but the other session does not see that and tries to update it. Yet another problem is locking. If two sessions enter records for the same DEPTNO, the second one had to wait to get the lock until the first one commits. This creates an all too obvious artificial row locking conflict.

All these, of course, created significant problems. We needed another solution. That was where advanced queue approach was considered for this use. Advanced queues are typically used for application development where one part of the processing is de-coupled from the others. However, the model may be used in various other situations; for example this one. In order to present the concepts as concisely as possible the same oversimplified example shown above will be used. This is much less complicated than the actual case, but the problems and the solutions are more or less in line with the actual solution and hopefully, the user will appreciate the design.

WHAT ARE ADVANCED QUEUES

Advanced Queue (AQ) is a queue-based system entirely within the database. A queue is simple in concept – it is like a pipeline. Anyone can place something in a queue that waits inside until someone picks it up from the queue. This is typically done in a first in first out basis. That someone who places information in the queue is called a *producer*; the information that goes into the queue is called a *message* and that someone who picks it from the queue is called a *consumer*. The message carries the useful data called a *payload*. A message is like an envelope with the payload as the letter. The act of placing payload in the queue is called *enqueuing* and that of retrieving is called *dequeuing*. There can be more than one producer and consumer for a queue. The producer and consumers can be same process too. A queue can have *subscribers*, processes that get message off the queue but that does not prevent other subscribers from dequeuing the same message. This is called a *multi-consumer publishers-subscriber* model queue. There can be *rules* defined that allows a certain subscribers to dequeue payloads from the queue, not all.

AQs can be *persistent* or *non-persistent*. The former type is not in memory but physically stored. In case of a database shutdown, the queue contents are preserved. This type of queue is also protected by the database; it can be backed up and recovered. The latter type of queue is in memory only; they disappear as soon as the instance is brought down. Needless to say, the performance is better in non-persistent queues.

Perhaps the single most important attribute of an AQ is the ability to participate in a transaction – i.e. the message is placed in the queue only when the overall transaction commits; and is never placed if the transaction is rolled back. This characteristic of AQ provides rich benefits for building a messaging system that is transaction-aware, or in other words, able to handle real world business processes.

So in effect, the AQ based system is like a huge delivery system. Being entirely within Oracle database, it's protected from failures and is transaction-aware. Each queue is associated with a database table called Queue Table (QT) that can hold more than one queue. Each QT can handle only one type of payload. The payload can be a simple varchar2 string or an Oracle Abstract Datatype (ADT) object to define the payload. The QT being a physical table can also have its storage attributes such as tablespace, initial extent, etc. defined.

A queue is created in a Queue Table. A queue called *exception queue* is always created automatically in a queue table. This is used to retain the messages that failed while dequeuing. The queue system is managed by two Oracle supplied packages named DBMS_AQ and DBMS_AQADM. In addition to the Oracle Enterprise Manager Console.

SOLUTION APPROACH

Coming back to the problem we have, the proposed solution involves building a queue with the triggers as producers that place the change data into the queue. On the other side only one consumer, a stored procedure dequeues the message and updates the summary table. Since the update is done by only one session, the problem of incorrect updates will never occur.

INITIAL DATABASE PREPARATION

The database has to set up with the following parameter and value in the init.ora file.

```
aq_tm_processes = 2
```

This sets up 2 AQ Time Manager processes that check the queues for messages for time related events like expiration, retry, etc. These can also be setup by ALTER SYSTEM. This parameter can have a value up to 10. The processes are started as background processes and named with a format QMN*n*, where n is the number of the monitor.

The users that will operate the queues need to be granted privileges associated with that. The following two lines establish that, provided the user is SCOTT. Connecting as SYS, issue these two statements.

```
GRANT EXECUTE ON DBMS_AQADM TO SCOTT;
GRANT EXECUTE ON DBMS_AQ TO SCOTT;
```

Setting Up the Working Procedures

The first task is to define a type that will be used in the change data transfer. This type is an Oracle Abstract Data Type (ADT) and will be used to define the payload of the message. The type is defined as follows. All statements from now on will be executed as the user SCOTT.

```
create or replace type dept_counts_type as object
(
   action_type   char(1),
   old_deptno   number(2),
   new_deptno   number(2)
)
```

Since we have to transmit the information on what happened and which deptno was affected, we have defined the old and the new department numbers and the action code which will be an one character string with values I, D or U, designating Insert, Delete or Update respectively. Next, we will build the procedure that will do the update on the summary table. One special consideration has to be made about this operation. Since the table is maintained by us and not by Oracle, what happens when someone truncates the EMP table, or even drop it? These events will not fire the trigger that captures the changes. Therefore, we need to introduce another event that truncates the DEPT_COUNTS tables when the EMP table is truncated or dropped. This can be done by passing a special action code, 'T' to the processing procedure.

```
create or replace procedure process_dept_counts
(  p_action          IN char,
   p_old_deptno      IN number,
   p_new_deptno      IN number)
as
   invalid_action  exception;
begin
   if (p_action = 'T') then
       execute immediate 'truncate table dept_counts';
   elsif (p_action = 'D') then
      update dept_counts
      set emp_count = emp_count - 1
      where deptno = p_old_deptno;
   elsif (p_action = 'I') then
      update dept_counts
      set emp_count = emp_count + 1
      where deptno = p_new_deptno;
      if (SQL%NOTFOUND) then
         insert into dept_counts
         values
         (p_new_deptno, 1);
      end if;
   elsif (p_action = 'U') then
      update dept_counts
      set emp_count = emp_count - 1
      where deptno = p_old_deptno;
      if (SQL%NOTFOUND) then
         null;
      end if;
      update dept_counts
      set emp_count = emp_count + 1
      where deptno = p_new_deptno;
      if (SQL%NOTFOUND) then
         insert into dept_counts
         values
        (p_new_deptno, 1);
      end if;
   else
      raise invalid_action;
   end if;
```

```
end;
```

Please note, to conserver space, I have not taken into account all possibilities while writing this code. For example, what happens when the EMP_COUNT becomes 0 in the DEPT_COUNTS table? The row should then be deleted. Similarly, other housekeeping tasks like exception handling have not been considered here. The example is presented to convey the general idea; the reader is expected to consider all aspects of the system when developing for a real system.

SETTING UP THE QUEUE SYSTEM

Now we will build the queue table, called `dept_counts_qt`.

```
begin
   DBMS_AQADM.CREATE_QUEUE_TABLE (
      queue_table        => 'DEPT_COUNTS_QT',
      queue_payload_type=> 'DEPT_COUNTS_TYPE',
      multiple_consumers=>FALSE,
      storage_clause      =>
'TABLESPACE USR INITRANS 10 STORAGE (FREELISTS 10 FREELIST GROUPS 2)',
      compatible          => '8.1');
end;
```

In this queue table, we will create the queue called `dept_counts_q`.

```
begin
   DBMS_AQADM.CREATE_QUEUE (
      queue_name =>'DEPT_COUNTS_Q',
      queue_table=>'DEPT_COUNTS_QT',
      max_retries=>'5',
      retry_delay=>'0');
      /* Now Start the Queue */
      dbms_aqadm.start_queue('DEPT_COUNTS_Q',TRUE,TRUE);
end;
```

Now we are ready to place the messages in the queue. We will create a generic procedure to enqueue into the queue `dept_counts_q`, so that it can be used by other applications as well. The procedure below will do that.

```
create or replace procedure enq_dept_counts_q
   (p_msg    in dept_counts_type)
as
   enq_opt  dbms_aq.enqueue_options_t;
   msg_prop dbms_aq.message_properties_t;
   msg_id   raw(16);
begin
   sys.dbms_aq.enqueue (
   'DEPT_COUNTS_Q',
   enq_opt,
   msg_prop,
   p_msg,
   msg_id);
end;
```

Similarly to dequeue the payload from the queue, we will use another generic procedure.

```
create or replace procedure deq_dept_counts_q
as
   deq_opt   dbms_aq.dequeue_options_t;
   msg_prop  dbms_aq.message_properties_t;
   payload   dept_counts_type;
   msgid     raw(16);
begin
   loop
      deq_opt.wait := dbms_aq.forever;
      deq_opt.navigation := dbms_aq.next_message;
      dbms_aq.dequeue(
         'DEPT_COUNTS_Q',
         deq_opt,
         msg_prop,
         payload,
         msgid
      );
      process_dept_counts(
         payload.action_type,
         payload.old_deptno,
         payload.new_deptno);
      commit;
   end loop;
end;
```

The final piece is to send the messages to the queue via a trigger. We can create a trigger on the EMP table as per the logic we discussed above.

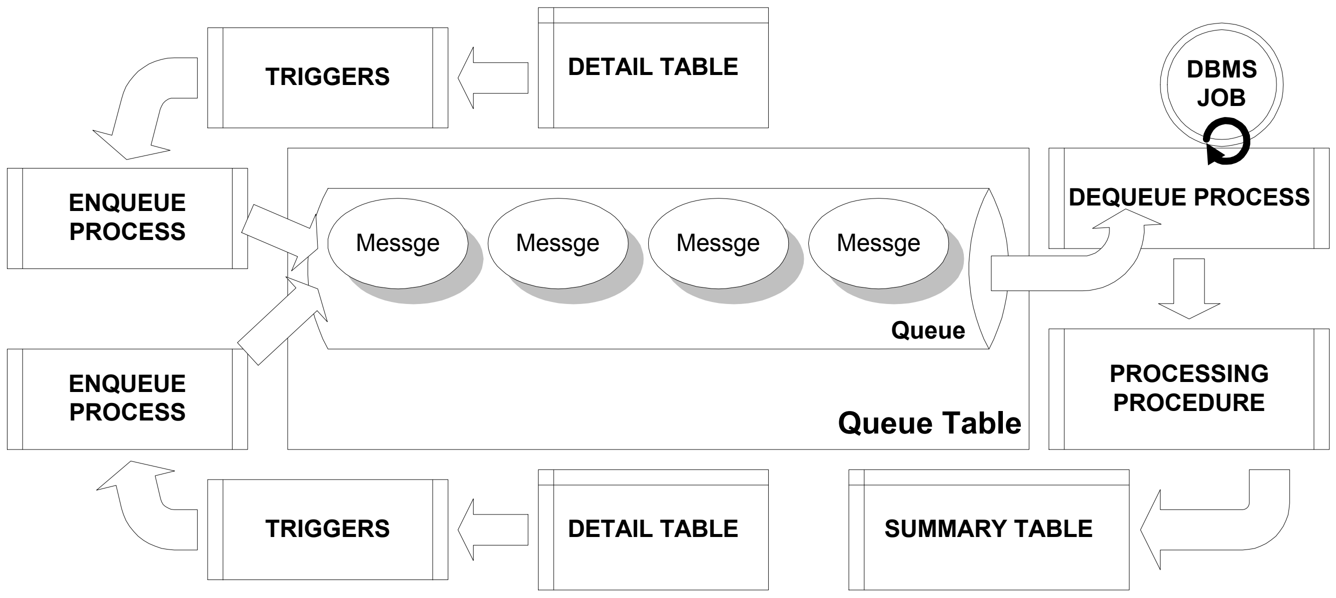
```
create or replace trigger tr_ar_iud_emp
after insert or delete or update on emp
for each row
declare
   l_action      char(1);
   l_old_deptno  number(2);
   l_new_deptno  number(2);
begin
   if (inserting) then
      l_action :='I';
      l_old_deptno := null;
      l_new_deptno := :new.deptno;
   elsif (deleting) then
      l_action :='D';
      l_old_deptno := :old.deptno;
      l_new_deptno := null;
   elsif (updating) then
      l_action :='U';
      l_old_deptno := :old.deptno;
      l_new_deptno := :new.deptno;
   else
      null;
   end if;
   enq_dept_counts_q(
      dept_counts_type(
         l_action,
         l_old_deptno,
         l_new_deptno));
end;
```

For the truncate or drop part, we need to code that logic in a database trigger.

```
create or replace trigger tr_trunc_or_drop_table
after drop or truncate on schema
declare
   ev   varchar2(200);
   ow   varchar2(200);
   ty   varchar2(200);
   ob   varchar2(200);
   errm varchar2(2000);
   l_msg   dept_counts_type :=
      dept_counts_type('T',null,null);
begin
   ow := ora_dict_obj_owner;
   ty := ora_dict_obj_type;
   ob := ora_dict_obj_name;
   if (   (ow = user) and
      (ty = 'TABLE') and
      (ob = 'EMP'))
   then
      enq_dept_counts_q (l_msg);
   end if;
end;
```

Now, we have a complete system to maintain the summary table DEPT_COUNTS from the EMP table. The procedure deq_dept_counts_q is run from the SQL prompt or in the background via a shell script and its runs in an infinite loop until it is killed. It continues to listen to the queue and processes the messages that pass through it. The idea can be extended to cover more than one source tables and even more than one summary tables. The queues can be extended to cover a variety of changes or a separate queue can be created for each type of payload. The former is needed when the changes need to be applied as one atomic transaction. To facilitate the process, the dequeue process can be kicked off from a database job through dbms_job, in which case the job will automatically start at the database startup; making administration even simpler.

The system, completely assembled is depicted in the figure..



IMPORTANT CONSIDERATION

The whole system is based on an asynchronous and decoupled transaction-processing model. Please note that since they are decoupled, the normal database properties like read consistency are not available. This may render the system infeasible in specific situations. For instance, in this example, if the source session inserts into EMP table but does not commit, it will not be able to see that record in DEPT_COUNTS table. This is unlike in a regular transaction processing system where the source will be able to see the record but not others. This important property must be carefully considered while applying the model to a real world system.

Administration

This system itself does not need any type of administration; however sometimes it may be worthwhile to peek under the hood to check the workings. The frequently requested procedure is to identify the number of messages in the queue at a present time. This can be done by the following query.

```
select count(*) from AQ$DEPT_COUNTS_QT where queue = 'DEPT_COUNTS_Q'
```

If the messages are not retrieved from the queue, check the exception queue, named after the queue table and the suffix _QE. If the queue table name is DEPT_COUNTS_QT, then exception queue is named `AQ$DEPT_COUNTS_QT_QE`.

Other Uses

As seen from the example, this AQ based system can be used to maintain any type of summary tables, not just in data warehousing environments but in practically in any type of application, OLTP, web based or a mixture of DSS and OLTP. The near real time performance of this system makes it ideal for several types of usage without placing undue burden on the system resources as well as making users happy with faster response. The faster response, down to seconds as opposed to hours opens up another possible use of this system – using it in normal applications replacing complicated queries. For instance, a banking application might use this to while displaying a customized message to the customer depending upon the various relationships of her with bank. Typically, this needs complicated queries and response time sometimes wane. Instead of adding additional indexes, which may not be feasible, the AQ system may be used to maintain a near real time summary table for the relationships, and the application can select from the table.

This example was intended to provide the concepts and workings of a functional AQ based system. Real life situations are much more complex but the inner working remain the same. The objective of the exercise was to introduce a relatively unknown usage of an Oracle tool to solve a well-known problem.

About the Author

Arup Nanda has been an Oracle DBA for more than ten years working in all aspects of database design and management such as performance tuning and disaster recovery planning. He is the founder of Proligence, Inc. (www.proligence.com), a New York area based Oracle database services firm providing extremely specialized services like setting up replication, parallel server, performance tuning and providing alternative solutions like this case here. He can be reached at arup@proligence.com.