

REAL-TIME DATA AUDITING WITH EXTENDED ORACLE CHANGE DATA CAPTURE

Scott Jia, Davis & Henderson Ltd.

DATA AUDITING CHALLENGES IN AN ENTERPRISE ENVIRONMENT

With ever strengthened security management guidance from industry, Oracle is building more and more security features, options into its database and provides separate products, like Audit Vault to bring companies more compliant to regulations such as SOX, PCI and HIPAA, etc.

The charm about Oracle Audit Vault is that, you can create policies to audit certain SQL statements, access to database objects, privilege usages, fine-grained audit conditions, and recover user activities including before/after values of a transaction from redo log data. This latter capacity makes it possible to keep a track of data change history, which is very desirable from an auditing's point of view.

Ever since the introduction of Oracle Change Data Capture (CDC) in 9i, companies with a more constraint budget are wondering: could it be possible to achieve similar functionality with this technology? Oracle Change Data Capture is designed to facilitate ETL process; it doesn't provide much information about user identity, specifically the end user identifier in a multi-tier environment. Nevertheless, Change Data Capture is greatly enhanced in 10g and 11g, it can now capture changes made by direct insert or nologging DML's, and its supported data type has been expanded to include LOB's in its asynchronous mode. Yet, the concern about missing user identity is still not addressed.

This paper describes a process we developed to extend change data capture to include additional application information, such as end user identifier, client info, etc. When deployed with Asynchronous Distributed HotLog mode, it provides following capacities which addressed many challenges for an enterprise wide audit solution,

1. Real-time change data capture.
2. Capture end user identifier in a multi-tier application environment.
3. Centralized staging environment.
4. Near real-time reporting with reassembled user activities and FGA audit trail.
5. Support multiple Oracle source databases.
6. Support different versions of Oracle databases running on different OS/hardware platforms.
7. Minimum performance impact on source systems.

To better understand these features, we'll briefly review the concept of Asynchronous Distributed HotLog mode in the next section. It also describes Asynchronous AutoLog archive mode, which takes advantage of Oracle downstream capture and minimize impact to source databases.

The major part of this paper focuses on how to extend change data capture, and provides best practices along the road. It demonstrates the ordering sequence to reassemble user activities and join criteria to accommodate FGA audit trail to change tables.

It briefly mentioned the processes to be automated when implementing the process in an enterprise environment.

CONCEPT OF ASYNCHRONOUS DISTIBUTED HOTLOG CHANGE DATA CAPTURE

Synchronous change data capture has been with Oracle since 9i. It is a trigger-based implementation and the capture process participate the transaction for which it is capturing, thus it imposes performance impact to the transaction. Asynchronous change data capture was made available with Oracle 10g in its enterprise edition. Contrary to synchronous mode, asynchronous mode utilizes log mining to recover user activity, and thus eliminate any impact to a transaction.

CHARMS OF ASYNCHRONOUS DISTRIBUTED HOTLOG MODE

Particularly, Asynchronous Distributed HotLog mode carries following features,

- Capture process invokes log mining process whenever redo data are written to online redo log. Recovered LCR's are enqueued and populated to staging database immediately. This is often close enough to meet most real-time auditing requirements.
- Apply process is typically running on a separate (staging) box, which further offset workload on source databases.
- You can configure multiple change sources, with each linked to its own source database. As a result, you can build a centralized staging and reporting box, which receives change data from multiple front databases.
- Most importantly, since the capture process resides in source database, its log mining and LCR enqueue process are totally transparent to the apply process running in staging database, this makes it capable to support source databases running with different Oracle versions on different OS/hardware platforms.

Well, there are some situations, e.g., in a supper busy OLTP Oracle database, it is more desirable to apply Asynchronous AutoLog Archive mode. This mode takes advantage of Oracle redo transport service and utilizes Oracle downstream capture process to eliminate capturing overhead from source database entirely. It also supports multiple change sources, but they have to be running in same Oracle version on same platform as the staging database. This is often proved to be too restrictive in an enterprise environment, and limited its adoption.

Given these considerations, we choose Asynchronous Distributed HotLog mode for our implementation. But the describe process in this paper also works with Asynchronous AutoLog archive mode when it is your choice.

EXTEND CHANGE DATA CAPTURE IN ASYNCHRONOUS DISTRIBUTED HOTLOG MODE

Look at the control columns of a change table, it hardly bears any information about end user identity, this situation hasn't been mitigated in Oracle 11g database either. Well as we understand, without this piece, we can't enforce accountability, and it makes any critical auditing essentially meaningless.

While there is no mist about the capacity for application to send this missing information to database, what we need is to develop a process to bring this information over to change tables.

The process starts with creating a statement trigger on each source table, the trigger captures desired application information set by code, and writes them into a centralized audit trail table in source database. This audit trail table is then replicated to staging database by a materialized view refresh process. A separate job is scheduled in staging database; it scans unprocessed rows in all change tables from same source database, and lookup their transaction id in the materialized view, it copies application information from the materialized view to change tables upon a match on transaction ID.

Above is a brief description of the process, details below with code examples. This process assumes a typical asynchronous distributed hotlog environment has already been configured, but for clarity reasons, some of these environment configuration codes are also provided when deemed necessary.

EXTEND CHANGE TABLES

Change tables by default consist of control columns and user columns specified when it is created, but it is supported to add additional columns thereafter, as shown below.

```
BEGIN
  DBMS_CDC_PUBLISH.CREATE_HOTLOG_CHANGE_SOURCE(
    change_source_name => 'ld',
    description => 'London source',
    source_database => 'ld_db');
END;
BEGIN
  DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
    change_set_name => 'ld_daily',
    description => 'change set for London product info',
    change_source_name => 'ld',
    stop_on_ddl => 'n');
END;
BEGIN
```

```

DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE(
  owner          => 'stg_cdcpub',
  change_table_name => 'ld_products',
  change_set_name  => 'ld_daily',
  source_schema   => 'sh',
  source_table    => 'products',
  column_type_list => 'prod_id number(6), prod_name varchar2(50)',
  capture_values  => 'both',
  rs_id           => 'y',
  row_id          => 'y',
  user_id         => 'n',
  timestamp       => 'y',
  object_id       => 'n',
  source_colmap   => 'n',
  target_colmap   => 'y',
  options_string  => 'tablespace cdc_ts');
END;
SQL>alter table ld_products add (
  client_identifier varchar2(64),client_info varchar2(64),
  module varchar2(48), action varchar2(32), machine varchar2(64),
  populated number(1));

```

These additional columns won't interrupt apply processes, they are simply ignored and are always set to null even if you've defined a no NULL default value.

As shown here, these additional columns all pertain to either user identity or their activities in application. Column 'machine' and 'program' are automatically populated by Oracle when the session is established. The 'populated' column is used to identify if this row has been processed, this will be explained in more details later.

Beware, the STOP_ON_DDL option in CREATE_CHANGE_SET statement is set to 'n' which is not a typical setting in an ETL process because a DDL operation on source table will often need further investigation to enforce data integrity, but it makes sense from a data auditing perspective where quality of irrelevant data is not an issue and the capture process should continue.

CAPTURE APPLICATION INFORMATION IN STATEMENT TRIGGER ON SOURCE TABLES

A statement trigger is defined on each source table. Such an example is given below.

```

CREATE OR REPLACE TRIGGER src_cdcpub.aud_products
AFTER INSERT OR UPDATE OR DELETE
ON SH.products
BEGIN
  /* detecting firing event and decide if proceed
  IF (not INSERTING) and
     (not DELETING) and
     (not (updating('PROD_ID') OR UPDATING('PROD_NAME'))))
  THEN
    return;
  END IF;
  /* write app info, current transaction ID to audit trail */
  INSERT INTO CDC_AUDIT_TRAIL(
    DB_NAME,
    TIME_STAMP,
    XID,
    OS_USER,
    SESS_USER,
    MACHINE,
    CLIENT_INFO,
    CLIENT_IDENTIFIER,
    MODULE,
    ACTION
  )

```

```

VALUES (
    upper(sys_context('USERENV','DB_UNIQUE_NAME')),
    sysdate,
    dbms_transaction.local_transaction_id(),
    sys_context('USERENV','OS_USER'),
    sys_context('USERENV','SESSION_USER'),
    sys_context('USERENV','HOST'),
    sys_context('USERENV','CLIENT_INFO'),
    sys_context('USERENV','CLIENT_IDENTIFIER'),
    SYS_CONTEXT('USERENV','MODULE'),
    sys_context('USERENV','ACTION')
);
Exception
  when others then
    -- to exit silently upon PK violation
    NULL;
END;

```

A primary key constraint is defined on (DB_NAME, XID) of CDC_AUDIT_TRAIL table.

A few points to mention,

1. The purpose of this trigger is to capture application information, which is identical for all activities within a same transaction. Thus, row level trigger is unnecessary. And because a statement level trigger fires only once for the statement, it impose less overhead than a row level trigger.

The trigger exits when it detects the updating columns are not of auditing interests. This further alleviates workload in a busy OLTP system where columns to be audited comprise a small portion of all changing columns.

2. The trigger intercepts current transaction ID and application information from USERENV namespace, and writes them to audit trail table. Keep in mind, the audit trail table has a primary key constraint on (DB_NAME, XID), thus any attempt to insert duplicated records will be captured by the exception handling part which simply fails the insert statement and exits silently. This ensures only the very first operation in a transaction can add a record to this audit trail table, all subsequent operations in the same transaction can't. This further decreases rows in this audit trail table, and assuages performance impact on source database.

SET APPLICATION INFORMATION IN CODE

It's application's call to set client identifier and other application specific information whenever it's necessary.

Application can invoke procedures in DBMS_APPLICATION_INFO and DBMS_SESSION packages to set application information, such as client_identifier, client_info, module, action, etc.

But a more efficient way to avoid a round trip between application and database is to call OCI interface to set attribute OCI_ATTR_CLIENT_INFO and OCI_ATTR_CLIENT_IDENTIFIER, or for Java, through setEndToEndMetrics method provided in oracle.jdbc.OracleConnection interface.

For applications utilizing identity management, enterprise user's identity is handled by framework. This information is also available in USERENV namespace.

POPULATE AUDIT TRAIL FROM SOURCE DATABASE TO STAGING DATABASE

This is achieved by creating a fast refreshable materialized view in staging database for the audit trail table in source database.

```

CREATE MATERIALIZED VIEW STG_CDCPUB.LD_AUDIT_TRAIL
BUILD IMMEDIATE
REFRESH FAST
START WITH SYSDATE
NEXT SYSDATE + 1/(24*60)
WITH PRIMARY KEY
AS
SELECT * FROM CDC_AUDIT_TRAIL@ld_db;

```

In this example, the materialized view is scheduled to be refreshed every 1 minute.

POPULATE EXTENDED COLUMNS IN CHANGE TABLES

So far we've brought over application information from source database to staging database, now we need to populate application information from LD_AUDIT_TRAIL to records of same transaction in all change tables from same source database.

This is better achieved by a separate job rather than a trigger on the materialized view, because in a busy system the matching process described below takes time to complete given it has to go over all change tables, this could significantly slow down the materialized view refresh process. The job performs following steps,

1. Look up XIDUSN\$, XIDSLT\$ and XIDSEQ\$ in control table for those records that haven't been processed (POPULATED column IS NULL),
2. Construct transaction ID by XIDUSN\$ || '.' || XIDSLT\$ || '.' || XIDSEQ\$,
3. Match it to XID column in the materialized view,
4. Upon such a match, update extended columns in change table with their corresponding columns in audit trail materialized view,
5. Set POPULATED to 1.
6. Repeat step 1 to 5 for all change tables.

The critical part of this process is to guarantee the uniqueness of transaction ID, which has been discussed and concluded at AskTom that transaction ID is practically will never repeated throughout a database's life span. Further, it is a document bug that states transaction ID returned by DBMS_TRANSACTION.LOCAL_TRANSACTION_ID is unique to an instance, in fact, as AskTom points out, it is unique across all instances in a RAC database.

Just as a lesson learned, we've previously used undocumented feature USERENV(' COMMITSCN ') to collect commit SCN of current transaction from the trigger on source tables, and join it to CSCN\$ column in change tables with a divergence to match records, it worked. But, once we confirmed the uniqueness of transaction ID, we've since simplified the matching criteria to use transaction ID. Besides, USERENV(' COMMITSCN ') is an undocumented feature, its behavior is subject to change which we disliked.

AUDIT REPORTING

Though direct access to change table is possible, it better handled in a publisher-subscriber fashion. This gives publisher more controls on which columns in change tables are visible to a specific subscriber. This is particularly important in a company where access to different parts of audit trail is split between different auditors.

REASSEMBLE TRANSACTIONS

Reassembling activities in the sequence as it happened in source database is achieved by ordering rows first on CSCN\$ and then on RSID\$ in all change tables. The charm about RSID\$ is that it is ordered at transaction level, that is it is not confined to a table. For example, suppose below is the event sequence you did in source database,

1. update row A in table X,
2. update row B in table Y,
3. update row C in table X

And say RSID\$ for step one is 800, it will then increase to 801 in step 2, and advance to 803 in step 3. From there we can track user activities across tables. Better, for an update operation, its before and after records in a change table share a same RSID\$ value, this makes it much easier to keep a track of data evolving history.

REPORTING WITH FINE-GRAINED AUDITING (FGA)

While change data capture keeps a track of data change history, unlike FGA which can define a value-based fine-tuned auditing condition, CDC doesn't provide such an intrusion detection capacity on offending DML's or queries.

From a reporting's point of view, it is beneficial if we could create a consolidated report covers records from both worlds. To achieve this, we need to identify TRANSACTIONID of a FGA record and join it to corresponding rows in change tables of same transaction.

Start from 10g, fortunately, there is a TRANSACTIONID column added to DBA_FGA_AUDIT_TRAIL table, this column is populated for all DML operations and also for the queries if they are executed within a transaction.

The column is defined as a RAW(8), you can derive the three pieces of a transaction ID as shown below,

```
XIDUSN$ := to_number(substr(transactionid,1,4), 'xxxx');
XIDSLT$ := to_number(substr(transactionid,5,4), 'xxxx');
XIDSEQ$ := to_number(substr(transactionid,9,8), 'xxxxxxxx');
```

You can then use these values to identify all relevant records in change tables of the same transaction.

Bewared, the TRANSACTIONID column is not populated for queries which are not part of a transaction, but this doesn't impose any concern because such a query doesn't logically belong to any transaction and thus there is no need to join its FGA audit trail to records in change tables.

PURGE AUDIT TRAIL

Managing the size of change tables is a little beyond the scope of this paper. Just as a brief, we found the addition of lower_bound to DBMS_CDC_SUBSCRIBE.PURGE_WINDOW in 11g makes a lot of sense. While in 10g, we actually refrain from using any provided purge process, instead, publisher manually maintain a retention window, and deletes obsolete records from change tables directly.

ENTERPRISE WIDE DEPLOYMENT AND MAINTENANCE

So far, we've manifested how to extend change data capture to include additional application information, and stitch a 10g FGA audit trail with change tables to complement an audit report. Deploy and maintain such a solution with a handful tables is one story, it's quite another if there are hundreds or even thousands of tables to be audited which is often the scheme in an enterprise backdrop.

In this circumstance, it is proven critical to automate following processes,

1. Add supplemental logging on source tables,
2. Create change tables,
3. Add additional columns to change tables,
4. Create triggers on source tables,
5. Create subscriptions,
6. Purge obsolete records in change tables and audit trail,
7. Report user activities on a given timestamp or SCN rang, or on a given transaction ID range.

Yet not all day-to-day maintenance efforts can be automated, for example, we still need to handle errors from capture process. These efforts have to be well estimated in project plan.

CONCLUSION

This paper demonstrated a process we developed to extend Oracle change data capture to bring over end user identifiers from application to change tables, and discussed the ordering and joining criteria to create a consolidated report with reassembled user activities and FGA audit trail. Its implementation with Asynchronous Distributed HotLog mode provided convincing advantages most desirable in an enterprise environment.

There is ongoing effort to improve the process, the PL/SQL packages and Perl scripts we developed for automation are still evolving.

And we look forward to future Oracle releases in a hope that many of our effort described here are built into change data capture.