Аудит изменений структуры БД, данных и протоколирование действий пользователя на примере СУБД Oracle

Даниил Шаров, к.п.н., доцент кафедры ИВТ, ОмГПУ, daniil.sharov@gmail.com

Уровень сложности: ★★☆

В данной статье рассматриваются различные подходы к протоколированию изменений структуры БД и данных, находящихся в БД. Используя возможности СУБД Oracle связанные с созданием DDL триггеров достигается аудит изменения структуры БД. Анализируются различные подходы к аудиту изменения данных и к выбору структуры журнала для сохранения изменений. Завершается статья примерами DDL триггера, пакета для создания журнала и протоколирования изменений.

Требуется знание SQL

Ключевые слова:Oracle, аудит изменений данных, аудит изменений структуры данных, журнализация, контроль действий пользователя, DDL trigger

Введение

Когда речь заходит об аудите в контексте СУБД, то в общем случае это может быть сохранение информации обо всем, что происходит в базе данных. Это может быть попытка авторизации пользователя, запросы к данным, запросы, не выполняющие изменений данных, и многое другое. В данной статье мы сосредоточимся только на аудите изменений структуры БД и хранящихся в ней данных. В целом аудит, журнализация или контроль изменений структуры и данных БД обычно решают следующие задачи [2]:

- узнать кто, когда и откуда производил изменения структуры БД и/или данных;
- ведение истории изменения структуры БД и/или данных;
- уведомление об изменениях структуры БД и/или данных;

В корпоративных информационных системах необходимо частично или полностью решать вышеприведенные задачи, особенно в информационных системах, которые предоставляют возможность изменять структуру БД.

В моей организации вопрос аудита изменения структуры и данных БД возник в процессе разработки информационной системы с изменяющейся структурой БД. Специфика приложения была такова, что позволяла пользователю создавать, изменять и удалять таблицы и поля в них. Конечно, использовался пользовательский интерфейс, адаптированный к конкретной предметной области, и пользователь мог не подозревать, что оказывает влияние на структуру БД. Структура БД менялась, менялись и данные, поэтому необходимо было обеспечить сохранение всех изменений, осуществляемых с данными, по крайней мере, за какой-то приемлемый период.

Обзор различных подходов к аудиту изменений в структуре БД и данных

Зачем придумывать свои средства аудита изменений, если существует возможность использовать стандартные средства аудита Oracle? Действительно, СУБД Oracle включает средства для контроля над различными действиями, осуществляемыми с БД. Рассмотрим лишь те средства, которые связаны с нашей основной задачей. По умолчанию в СУБД Oracle аудит отключен. Oracle не поставляется с какими-либо стандартными установками аудита и отчетами для анализа журнала аудита. Используя стандартные команды аудита, можно контролировать все системные и пользовательские операции доступа к любым таблицам или представлениям БД. Используя стандартные средства аудита, можно получать информацию о командах, которые приводили к изменению данных, но анализ, и тем более какие-либо модификации, в журналах аудита затруднительны. К тому же стандартные команды аудита не разрешают контролировать операции на уровне строк [3]. Поэтому мы для решения своей задачи вынуждены рассматривать различные подходы к детализированному аудиту и реализовать свой вариант.

Изменение структуры БД

Начнем с вопроса, как отследить изменение структуры таблиц в определенной схеме данных Oracle? К сожалению, для решения данной задачи возможностей не так уж много [4, 5, 6]. Основная идея — это создание триггера, кото-

рый позволяет отловить событие, связанное с изменением структуры БД [6]. Такие триггеры в Oracle появились, начиная с версии 8.1.6, и были существенно расширены в следующих версиях.

Примечание:

Все эксперименты проводятся на СУБД Oracle 10g XE для Windows, который можно бесплатно скачать с официального сайта Oracle: http://www.oracle.com/technology/software/products/database/xe/htdocs/102xewinsoft.html.

Триггер для отслеживания изменений в структуре какой-либо определенной схемы имеет вид, представленный в Листинге 1.

Листинг 1. Простой код для создания DDL-триггера.

```
create or replace trigger audit_schema_object
create or alter or drop on SCHEMA declare
begin
  -- Тело триггера
end;
```

Примечание:

DDL (Data Definition Language, язык определения данных) — это подмножество SQL, используемое для определения и модификации различных структур данных.

В теле триггера мы можем попытаться узнать, что за событие спровоцировало запуск триггера:

```
select ora_sysevent into l_sysevent from dual;
```

Название события, находящееся в ora_sysevent, помещается в l_sysevent — строковую переменную. Значением данной переменной является 'CREATE', 'ALTER' или 'DROP', в зависимости от типа операции. Чтобы узнать, какой объект подвергался воздействию, достаточно обратиться к ora_dict_obj_name в теле триггера. Тип объекта определяется через обращение к ora_dict_obj_type, и его значениями могут быть, например, 'TABLE' или 'COLUMN'. Владелец объекта определяется обращением к ora_dict_obj_owner.

Обратившись в теле триггера к представлению v$open_cursor при событии ALTER, мы можем получить исходную строчку кода (sql_text), которая привела к возникновению данного события. Полученную строчку кода, представленную в sql_text, можно разбирать в теле триггера и извлекать полезную информацию, например, для получения кода команды типа ALTER (Листинг 2).

Листинг 2. Запрос в DDL-триггере для получения кода команды на ALTER.

```
select ora_sysevent, ora_dict_obj_owner, ora_dict_obj_name, sql_text
  from v$open_cursor
  where upper(sql_text) like 'ALTER%' || ora_dict_obj_name || '%'
        and sid = (select sid from v$session
                   where audsid=userenv('sessionid'));
```

Рассмотрим простой пример, когда необходимо в таблице log фиксировать событие, объект, пользователя, осуществившего операцию (добавление, изменение, удаление), и код команды для alter. Создадим такую таблицу:

Листинг 3. Создание таблицы log.

```
create table log (
  operation varchar2(25), -- название события
  owner varchar2(25), -- владелец объекта
  nameobject varchar2(25), -- название объекта
  typeobject varchar2(25),-- тип объекта
  text varchar2(60), -- код, который привел к возникновению события
  username varchar2(30) default USER -- имя пользователя
  datechange date default sysdate -- дата и время изменения
);
```

В поле username по умолчанию будет добавляться строка, содержащая имя текущего пользователя СУБД Oracle. Схему, в которой мы будем работать, назовем METAXPO, а триггер, контролирующий модификацию структуры БД, назовем ddl_trigger. Итак, триггер ddl_trigger будет реагировать на все события, происходящие в структуре БД: create, alter, drop. Код триггера приведен в Листинге 4.

Листинг 4. Код триггера ddl_trigger для конкретной схемы.

```
create or replace trigger ddl_trigger
  after create or alter or drop on SCHEMA
declare
  l_sysevent varchar2(25);
  l_text     varchar2(1000);
begin
  if ora_dict_obj_owner <> 'METAXPO' then
     -- если схема не METAXPO, то проигнорируем
     return;
  end if;
   select ora_sysevent into l_sysevent from dual; -- название события

  if (l_sysevent = 'CREATE' OR l_sysevent = 'DROP') then -- Создание или удаление
объекта
    insert into log -- добавление записи в таблицу log
      (operation, owner, nameobject, typeobject)
       select ora_sysevent, -- наименование события
         ora_dict_obj_owner, -- владелец объекта
         ora_dict_obj_name, -- название объекта
         ora_dict_obj_type -- тип объекта
         from dual;
  end if;

  if (l_sysevent = 'ALTER') then -- Изменение объекта
    insert into log -- добавление записи в таблицу log
      (operation, owner, nameobject, typeobject, text)
       select ora_sysevent,
         ora_dict_obj_owner,
         ora_dict_obj_name,
         ora_dict_obj_type,
         sql_text -- исходный текст команды
         from v$open_cursor v
         where upper(sql_text) like 'ALTER%' || ora_dict_obj_name || '%'
               and sid = (select sid from v$session
                          where audsid=userenv('sessionid'));
  end if;
end;
```

В запросе на извлечение данных об изменении таблицы используется представление v$session, в котором содержится информация об установленных соединениях. Например, кто и откуда подключен, какой статус у подключения, название машины, с которой произошло подключение и т.д. Извлекая данные из этого представления, мы их фильтруем по идентификатору сессии (соединения), который можем получить для текущего сеанса, обратившись к userenv('sessionid'). Кстати, чтобы создать триггер в схеме METAXPO, приведенный в Листинге 4, необходимо разрешить пользователю METAXPO доступ к системным представлениям v$open_cursor и v$session. Для этого необходимо, находясь под пользователем sys (as SYSDBA), выполнить команды:

```
grant select on sys.v_$session to metaxpo;
grant select on sys.v_$open_cursor to metaxpo;
```

Вышеприведенные команды разрешают пользователю METAXPO доступ к объектам схемы sys: v$open_cursor и v$session. Теперь пользователь METAXPO сможет создать триггер ddl_trigger, а при создании, модификации и удалении объектов схемы, например, таблиц (нас интересуют в основном таблицы и поля), в таблицу log будут добавляться соответствующие записи. Чтобы выполнять логику триггера только для таблиц, можно проигнорировать объекты других типов, добавив в начало тела триггера ddl_trigger строчки кода:

```
if ora_dict_obj_type <> 'TABLE' then
  return;
end if;
```

В вышеприведенном коде (Листинг 4) есть небольшой нюанс: выполнение операторов create, alter и drop в схеме METAXPO каким-либо другим привилегированным пользователем (отличным от METAXPO) не приведет к срабатыванию триггера. Чтобы триггер не был ограничен уровнем схемы, нужно создавать триггер уровня базы данных (database level ddl trigger), а уже в теле этого триггера фильтровать выполнение операторов DDL, связанных с интересующей нас схемой. В таком случае строчка в заголовке триггера, определяющая его уровень, будет изменена с:

```
after create or alter or drop on SCHEMA
```

на

```
after create or alter or drop on database
```

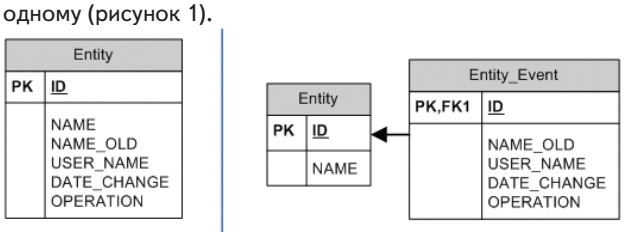
Отметим, что начиная с версии 9i, в СУБД Oracle появилась функция ora_sql_text, выдающая текст оператора, вызвавшего срабатывание триггера [6]. Использование этой функции намного проще и изящней, чем обращение к v$open_cursor для большинства задач. Более подробную информацию о доступных событиях и функциях СУБД Oracle при модификации структуры БД можно получить на официальном сайте Oracle [5].

Изменение данных

Итак, у нас существует возможность реагировать на изменение структуры БД и определять, что же было создано, изменено или удалено. Теперь перейдем к вопросу: как определить, какие данные в таблицах подвергаются изменению, и что это за изменения (добавление, обновление или удаление)? Используя триггеры, мы можем фиксировать не только факт изменения данных, но и получить доступ к информации, что это за данные, и какой модификации они подвергаются.

Здесь мы не будем рассматривать аудит выполнения операций чтения данных, запросов, которые не затрагивают ни одной записи. К тому же нас интересует не столько факт изменения, сколько накопление реальных данных, которые связаны с этим изменением (вспомним пример с тарифами, приведенный в начале статьи). К аудиту изменения данных нужно подходить с осторожностью, так как необдуманное и чрезмерное его использование может привести к существенному снижению производительности. Рассмотрим несколько вариантов создания аудита изменения данных. Выбор варианта зависит от требований, которые предъявляются к истории изменения данных [2].

Если нет необходимости хранить всю историю изменений, а достаточно сведений о последних изменениях в данных, то можно обойтись добавлением дополнительных полей (например: поля для хранения измененных данных, имени пользователя, даты и времени осуществления операции, типа операции (добавление, обновление) и т.д.) непосредственно в рабочей таблице или в отдельной таблице, которая связана с основной таблицей отношением один-к-одному (рисунок 1).



А Б

Рисунок 1. Структура таблицы для хранения последних изменений. Вариант А предполагает хранение данных в рабочей таблице, в варианте Б создается дополнительная таблица.

В поле NAME_OLD (Рис. 1) записывается предыдущее значение поля NAME, если оно было изменено. В поля USER_NAME и DATE_CHANGE записывается имя пользователя и дата/время последнего изменения, соответственно. В поле OPERATION записывается тип операции, например, значение 0 означает, что запись добавлена, а 1 — что запись изменена.

К недостаткам данного решения можно отнести то, что операцию удаления записи отследить невозможно. Обойти это ограничение можно, если физически не удалять записи, а помечать их как удаленные, добавив соответствующее поле-признак. Конечно, рабочая таблица при этом разрастается, и нужно ее периодически очищать от устаревших записей, помеченных на удаление.

Упрощенный вариант, когда нужно фиксировать только факт выполнения операции с данными, предполагает создание отдельной таблицы на каждую рабочую таблицу (рисунок 2).

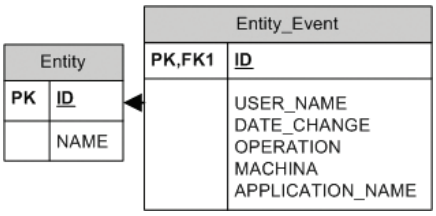


Рисунок 2. Структура таблицы для хранения только факта выполнения изменений.

В полях COMPUTER записывается имя компьютера, с которого осуществлена модификация данных, а в поле APPLICATION_NAME записывается название приложения, выполняющего изменение данных.

С помощью триггеров фиксируем факт выполнения операции. Реализовать всю логику можно в одном триггере, который будет реагировать на три операции, или в трех триггерах, по одному на каждую операцию (insert, update, delete).

При аудите изменений данных в нескольких таблицах необходимо помнить о порядке выполнения этих операций, т.к. если таблицы связаны между собой внешним ключом, это может привести к ошибкам в будущем (добавляется дочерняя запись, когда родительская еще не создана).

Если целью аудита является фиксация фактов изменений данных в определенных полях, то логика в триггерах может немного усложниться, но производительность в целом повысится, а размер таблицы для хранения изменений (журнала) снизится.

Рассмотрим, какую же структуру журнала лучше всего выбрать, и в каких случаях лучше реализовывать все в одной таблице, а в каком случае — в нескольких.

Какую структуру журнала выбрать?

Заранее оговорим, что не существует какой-то оптимальной структуры для хранения изменений в данных (журнала). Требования каждой конкретной задачи предполагают определенные ограничения и возможности, используя которые, можно реализовать наиболее подходящий вариант. Здесь мы рассмотрим несколько вариантов структур журнала. Выбор варианта зависит от того, какие критерии являются наиболее важными [2]:

- скорость функционирования системы с журналом изменений;
- объем журнала изменений и БД;
- удобство извлечения информации из журнала изменений;
- скорость извлечения информации из журнала изменений, его очистка и т.д.;

- использование блокировок при работе в многопользовательском режиме, при записи данных аудита в разные таблицы;
- универсальность решения;
- удобство сопровождения и масштабируемость решения (количество триггеров, их сложность, объем, способ формирования, исправление ошибок).

Ознакомившись с критериями, влияющими на выбор структуры журнала, перейдем к вариантам реализации самого журнала.

Случай, когда мы храним изменения в тех же таблицах, только с дополнительными полями, представлен на Рис. 1 (А). Когда запись удаляется, значение в числовом поле OPERATION меняем на 2. При этом все записи остаются в таблице, но удаляемые помечаются. В дальнейшем, ориентируясь на дату изменения и признак удаления, можно очищать таблицу от устаревших данных.

Другим вариантом аудита всех изменений в рамках определенной схемы БД является создание одной таблицы Audit (рисунок 3), в которую и записываются сведения об изменениях.

При реализации аудита нужно учитывать, что структура рабочих таблиц может различаться. Поэтому для записи значений придется выбрать тип поля, позволяющий сохранить значение любого типа. В ряде случаев это может быть строка (varchar2) или CLOB, а возможно, и BLOB (типы, присутствующие в СУБД Oracle).

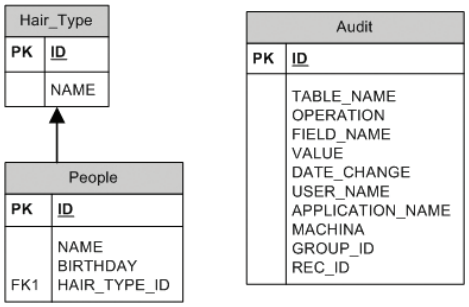


Рисунок 3. Структура таблицы (Audit) для хранения изменений в рабочих таблицах схемы данных с одним полем универсального типа.

Кроме полей, хранящих сведения о том, кто, когда, как и откуда изменял данные, в таблице Audit (рисунок 3) имеются поля TABLE_NAME (название таблицы, в которой происходит модификация данных), FILED_NAME (название поля, значение которого подверглось модификации), VALUE (новое значение) *(Мягко говоря, не самое эффективное решение —прим.ред.)*. В примере поле VALUE может иметь тип VARCHAR2, т.к. такой тип позволяет хранить и значение типа строка, и значение типа дата и время. Поле REC_ID содержит значение первичного ключа той таблицы, в которой произошли изменения. Если в рабочей таблице запись удаляется, то соответствующая запись остается в таблице Audit. Если запрос затрагивает сразу нескольких полей, используется формирование одинакового значения для поля GROUP_ID всех создаваемых записей — это позволит в дальнейшем определить, что запрос был один.

Например, выполнение запросов:

```
insert into People(name) values('Имя')
и
update People set name='Новое имя', birthday= sysdate where id = 1,
```

для структуры, представленной на Рис. 3, приведет к созданию следующих записей в таблице1.

Значение в поле OPERATION может принимать значения: 0 — добавление, 1- обновление, 2 — удаление.

К преимуществам данного подхода можно отнести то, что при добавлении в структуру новых таблиц не требуется производить много изменений в коде, триггеры для новой таблицы создаются аналогично уже существующим. В случае выполнения UPDATE в таблице Audit хранятся только значения измененных полей, что снижает размер таблицы *(на каждое поле в данном решении приходится столько дополнительной информации, записанной неэффективным образом, что об экономии здесь говорить не приходится — прим.ред.)*.

Недостатком является использование универсального типа данных (например, CLOB или BLOB), что в дальнейшем приводит к множеству явных и неявных приведений типов в триггерах и запросах к таблице Audit. При увеличении размера таблицы это, безусловно, ведет к снижению производительности. Отображение одной операции в виде нескольких записей в таблице Audit может привести к дополнительным сложностям при чтении изменений и их последующей обработке.

Проблему, связанную с одним универсальным полем, можно попытаться решить. Таблица Audit может содержать по одному полю для каждого типа, используемого в рабочих таблицах схемы данных (рисунок 4). Такая структура таблицы позволит избавиться от явных и неявных преобразований, что увеличит скорость при выполнении запросов, но увеличит и размер таблицы *(производительность СУБД часто намного больше зависит от того, умещаются ли данные в оперативную память сервера или нет, так что увеличение объема хранимых данных с большой вероятностью приведет к замедлению работы — прим.ред.)*.

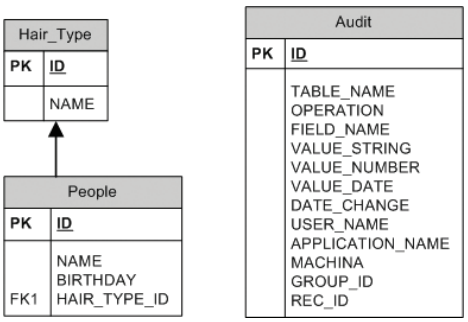


Рисунок 4. Структура таблицы (Audit) для хранения изменений в рабочих таблицах схемы данных с одним полем на каждый тип, присутствующий в рабочих таблицах схемы.

Следующим вариантом является создание копий рабочих таблиц для записи в них изменяемых данных, т.е. для каждой рабочей таблицы создается ее клон со всеми основными полями и с рядом дополнительных полей для хранения служебной информации (рисунок 5).

Аудит_4-Pics/pic_5.png

Таблица 1. Данные таблицы Audit после выполнения запросов

ID	TABLE_NAME	FIELD_NAME	OPERATION	VALUE	REC_ID	GROUP_ID	Служ. поля
1	People	NAME	0	Имя	1	1	
2	People	NAME	1	Новое имя	1	2	
3	People	BIRTHDAY	1	15.02.2010	1	2	

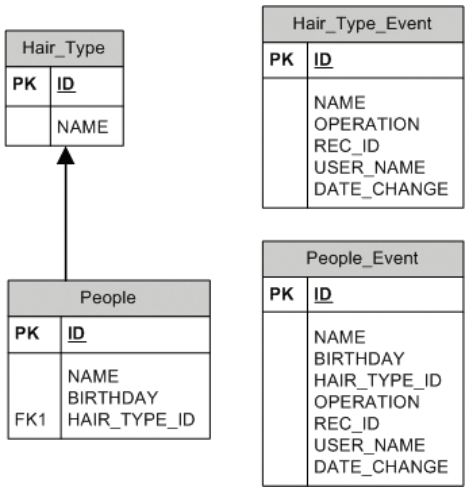


Рисунок 5. Структура таблиц-клонов для хранения изменений в рабочих таблицах схемы данных.

К преимуществам такого решения можно отнести высокую скорость выполнения запросов, т.к. нет преобразований типов. Повышается удобство извлечения данных из таких таблиц для дальнейшего анализа. Недостатком является значительное увеличение количества таблиц в схеме данных. Кроме этого, возможны проблемы с количеством полей в таблице-клоне, если в рабочей таблице количество полей находится на грани допустимого для СУБД. К недостаткам можно отнести также тот факт, что при изменении значения хотя бы одного из полей рабочей таблицы в таблицу-клон придется добавить полноценную запись со всей служебной информацией.

В данной статье мы рассматриваем только вариант, когда все контролируется средствами СУБД (с помощью триггеров). В качестве альтернативы при разработке информационной системы можно рассмотреть возможность аудита на клиенте. Подобное решение позволит снизить нагрузку на СУБД, а в ряде случае увеличит гибкость системы *(остается вопрос, будет ли такое решение безопасным — прим.ред.)*.

Реализация аудита изменений в структуре БД и данных

После обзора возможностей и подходов к контролю изменений структуры и данных БД приступим к созданию своего решения. Остановимся на решении выполнять весь контроль над изменением структуры БД и данных на сервере. Нам необходимо будет разработать триггер уровня базы данных, который будет отслеживать все модификации структуры и соответственно реагировать на эти изменения. Какой же вариант аудита нам выбрать в рамках нашей задачи, когда необходимо «прозрачно» обращаться к актуальным данным и к «архивным» данным? Под прозрачностью будем понимать простоту и скорость доступа к «архивным» данным. Архивными данными, как вы вероятно уже догадались, являются данные, которые представляют собой цепочку изменений данных в рабочих таблицах.

Мы решили остановиться на создании для каждой рабочей таблицы, для которой необходимо вести аудит, специальной таблицы в которую записывать все изменения. При таком варианте в рабочей таблице всегда будут только актуальные данные, и ее участие в запросах не приведет к снижению производительности. Для того чтобы в триггере можно было различать где происходят изменения в рабочей таблице или в таблице-клоне, мы введем правила именования таблиц. Пусть наименование таблицы-клона будет аналогичным названию соответствующей рабочей таблицы, но с постфиксом _Event, т.е. для рабочей таблицы Table1 таблица клон будет называться Table1_Event.

Перейдем к структуре таблицы-клона. Введем в структуру такой таблицы искусственный первичный ключ, который назовем ID. Также добавим обязательные служебные поля:

- UserName — имя пользователя;
- DateChange — дата изменения;
- Operation — тип произведенной операции;
- RecId — ссылка на первичный ключ соответствующей записи рабочей таблицы.

В рабочих таблицах не должно быть полей с такими именами. Если это невозможно, придется добавлять логику по переименованию названий полей, но реализация подобной логики не должна вызывать особой сложности. Постараемся реализовать всю основную логику создания таблиц-клонов в пакете, а функции, реализованные в данном пакете, будем вызывать из DDL-триггера.

Создание триггера DDL на модификацию структуры БД

В примере триггера audit_db_object (Листинг 5) мы ограничимся аудитом создания таблицы, проигнорируем обращение к объектам схемы, отличной от METAXPO, и не будем реагировать на обращения к таблицам, в конце имени которых встречается _EVENT или _INDEPENDENT. Если не нужно, чтобы триггер запускал механизм создания таблицы-клона, в имя таблицы будет добавляться постфикс _INDEPENDENT.

Листинг 5. Код триггера audit_db_object, в котором вызываются функции пакета metadata_pkg.

```
create or replace trigger audit_db_object
  after create or alter or drop on database
declare
  sqlCreateScript varchar2(2000);
  const_schema constant VARCHAR2(10) := 'METAXPO';
BEGIN
  -- Контроль: не реагировать если модифицируется структура КЛОНА
  if ora_dict_obj_owner <> const_schema then
    return;
  end if;

  -- Игнорируем таблицы, которые создаются для сбора событий,
  --  и таблицы помеченные, как независимые '_INDEPENDENT'
  if INSTR(upper(ora_dict_obj_name), '_EVENT') > 0
    OR INSTR(upper(ora_dict_obj_name), '_INDEPENDENT') > 0
  then
    return;
  end if;

  -- Создание таблицы
  if (ora_dict_obj_type = 'TABLE' AND ora_sysevent = 'CREATE') then
    -- Создается клон созданной таблицы
    metadata_pkg.CreateCloneTable(ora_dict_obj_name);
  end if;
END;
```

Триггер выглядит довольно просто, но в нем присутствует строчка, которая вызывает операцию создания таблицы-клона. Перейдем к рассмотрению содержимого пакета metadata_pkg.

Пакет для создания таблицы-клона по аналогии с рабочей таблицей

Описание пакета

Описание пакета metadata_pkg представлено в листинге 6. Нам доступна процедура CreateCloneTable, которая получает на вход в качестве аргумента имя рабочей таблицы, и ничего больше.

Листинг 6. Заголовок пакета metadata_pkg

```
create or replace package metadata_pkg
AUTHID DEFINER
is
  PROCEDURE CreateCloneTable(tableName in varchar2);
end metadata_pkg;
```

Тело пакета

При создании данного пакета нужно решить несколько вопросов. Создание таблицы-клона требует наличия искусственного первичного ключа ID, а его значения нужно генерировать автоматически. Для этого потребуется создать последовательность (sequence) и триггер (before), в котором будет записываться значение первичного ключа при добавлении записи.

Выполнение DDL-команд из нашего триггера (листинг 5) завершится неудачей. Попытка использовать динамический SQL с DDL-командами посредством вызова EXECUTE IMMEDIATE (позволяет выполнить SQL-команду из строки) также потерпит фиаско. DDL-команды включают в себя операцию подтверждения (commit) после выполнения, а подобные операции в триггерах запрещены. Придется искать обходные пути. Будем создавать задания (JOB) в процедурах, реализованных в пакете metadata_pkg, используя dbms_job.submit. Но одно задание может выполнить только одну команду, поэтому для того, чтобы добиться последовательного выполнения команд, нужно создать задание, в котором будет использоваться динамический sql. В процедуре CreateCloneTable (листинг 7) последовательно создается строка, в которой используется EXECUTE IMMEDIATE для создания таблицы-клона, последовательности, триггера и т.д.

Листинг 7. Тело пакета metadata_pkg

```
create or replace package body metadata_pkg
  is
  -- Префикс имени последовательности
  cntPrefixSq CONSTANT VARCHAR2(3):='sq_';
  -- Префикс имени триггера
  cntPrefixTrigger CONSTANT VARCHAR2(3):='tr_';
  -- Постфикс имени таблицы-клона
  cntPostfixTable CONSTANT VARCHAR2(6):='_Event';

  /* Обрамляет скрипт командой для выполнения динамического sql */
  function CreateScriptForImmedate (script in varchar2) return varchar2
  is
  begin
    return 'execute immediate ''' || script || '''';';
  end;

  /* Обрамляет скрипт командой для выполнения динамического sql */
  function CreateScriptForImmedateInner(script in varchar2) return varchar2
  is
  begin
    return 'execute immediate ''''' || script || '''''';';
  end;

  /* Обрамляет скрипт блоком для пакетного выполнения динамического sql */
  function CreateScriptForBegin (script in varchar2) return varchar2
  is
  begin
    return 'begin ' || chr(10) || script || chr(10) || ' end;';
  end;

  /* «Создает» (возвращает скрипт для создания) последовательность */
  function CreateSequence(tablename in varchar2) return varchar2
    is
  begin
    return 'create sequence ' || '»' || cntPrefixSq || tablename
           || '» ' || 'minvalue 1
    maxvalue 999999999999999999999999999
    start with 1
    increment by 1
    cache 20';
end;

/* «Создает» (возвращает скрипт для создания) «болванки» таблицы */
function CreateTable (tablename in varchar2) return varchar2
is
begin
 return CreateScriptForImmedateInner(' create table «' || tablename || '» '
        || '(ID NUMBER not null)') || chr(10)
        || CreateScriptForImmedateInner('alter table «' || tablename
        || '» add primary key (ID)');
end;

-- «Создает» (возвращает скрипт для создания) полей */
function CreateColumn (tablename in varchar2, columnName in varchar2,
          dateType in varchar2, charLength in number, nullable in varchar2,
          defaultValue in varchar2) return varchar2
is
 result varchar2(5000);
begin
  result := 'alter table «' || tablename || '» add «' || columnName || '» '
          || dateType || ' ';

 if charLength > 0 then
   result := result || '(' || charLength || ' CHAR)';
 end if;

  if nullable = 'N' then
    result := result || ' not null';
  end if;

  if (length(defaultValue)>0) then
    result := result || ' ' || defaultValue;
  end if;

  return result;
END;

-- «Создает» (возвращает скрипт для создания) триггера */
function CreateTrigger (tablename in varchar2) return varchar2
is
begin
  return 'create or replace trigger «' || cntPrefixTrigger || tablename
         || '_Ins» before insert on «' || tablename || '»
for each row
declare
begin
  if (inserting) then
    select ' || '»' || cntPrefixSq || tablename
     || '».nextval into :new.oid from dual;
  end if;
end «' || cntPrefixTrigger || tablename || '_Ins»;';
END;

/* Создание клона указанной таблицы с системными полей, последовательностью
   и триггером */
procedure CreateCloneTable(tablename in varchar2) is

  sqlCreateScriptSequence varchar2(1000);
  sqlCreateScriptTable varchar2(1000);
  sqlCreateScriptStructureTable varchar(10000);
  sqlCreateScriptTrigger varchar2(3000);
  tableNameClone varchar2(30);
  job_no_out number;
begin
  tableNameClone := tablename || cntPostfixTable;
  sqlCreateScriptSequence := CreateScriptForImmedateInner(
                             CreateSequence(tableNameClone));
  sqlCreateScriptTable := CreateTable(tableNameClone);
  sqlCreateScriptStructureTable := '';

  for rec in (select C.COLUMN_NAME,
                     sysdate,
                     0,
                     C.TABLE_NAME,
                     C.DATA_TYPE,
```

```
                    C.CHAR_LENGTH,
                    C.NULLABLE
                from ALL_TAB_COLUMNS C
                where UPPER(C.OWNER) = UPPER(SYS_CONTEXT(
                                    'USERENV', 'CURRENT_SCHEMA'))
                    and trim(C.TABLE_NAME) = trim(tablename))
  loop
    if (rec.COLUMN_NAME<>'OID' and rec.COLUMN_NAME<>'ID') then
      sqlCreateScriptStructureTable := sqlCreateScriptStructureTable
        || CreateScriptForImmedateInner(CreateColumn(tableNameClone,
             rec.COLUMN_NAME, rec.DATA_TYPE, rec.char_length,
             rec.nullable, '')) || chr(10);
    end if;
  end loop;

  -- Дополнительная структура (полей)
  sqlCreateScriptStructureTable := sqlCreateScriptStructureTable
    || CreateScriptForImmedateInner(CreateColumn (tableNameClone,
        'UserName', 'varchar2', 30, 'Y', 'default USER')) || chr(10);
  sqlCreateScriptStructureTable := sqlCreateScriptStructureTable
    || CreateScriptForImmedateInner(CreateColumn (tableNameClone, 'RecId',
       'number', 0, 'Y', '')) || chr(10);
  sqlCreateScriptStructureTable := sqlCreateScriptStructureTable
    || CreateScriptForImmedateInner(CreateColumn (tableNameClone,
         'DateChange', 'date', 0, 'Y', 'default sysdate')) || chr(10);
  sqlCreateScriptStructureTable := sqlCreateScriptStructureTable
    || CreateScriptForImmedateInner(CreateColumn (tableNameClone,
         'Operation', 'number', 0, 'N', '')) || chr(10);
  sqlCreateScriptTrigger := CreateScriptForImmedateInner(
                              CreateTrigger(tableNameClone));

  dbms_job.submit(job => job_no_out, what =>
    'begin ' || chr(10)
      || CreateScriptForImmedate(CreateScriptForBegin(
           sqlCreateScriptSequence || chr(10)
           || sqlCreateScriptTable || chr(10)
           || sqlCreateScriptStructureTable || chr(10)
           || sqlCreateScriptTrigger)
         )
      || chr(10) || 'end;',
    next_date => sysdate +(1/24/60/60),
    no_parse => TRUE);
 end;
end metadata_pkg;
```

Аудит модификации (добавление, изменение, удаление) данных

Создаем триггер after (после), отслеживающий операции insert, update и delete в рабочей таблице. В этих триггерах должна осуществляться вставка данных в клонируемую таблицу с постфиксом _Event. Команда для создания таблицы TBL_1 приведена в Листинге 8.

Листинг 8. Команда создания таблицы TBL_1.

```
create table TBL_1
(
  OID  NUMBER        not null, -- Первичный ключ
  NAME VARCHAR2(30),
  IDX  NUMBER
)
```

Чтобы фиксировать (записывать) изменения, добавим триггер (Листинг 9).

Листинг 9. Триггер на таблицу TBL_1 для фиксирования операций insert, update, delete.

```
create or replace trigger tr_TBL1_After_Modify
  after insert or update or delete on TBL_1
  for each row
declare
begin
  if (inserting) then -- Добавлена
    insert into «TBL_1_Event» (NAME, IDX, «RecId», «Operation»)
      values(:new.name, :new.idx, :new.oid, 0);
  end if;
  if (updating) then -- Изменение
    insert into «TBL_1_Event» (NAME, IDX, «RecId», «Operation»)
      values(:new.name, :new.idx, :new.oid, 1);
  end if;

  if (deleting) then -- Удаление
    insert into «TBL_1_Event» (NAME, IDX, «RecId», «Operation»)
      values(:old.name, :old.idx, :old.oid, 2);
  end if;
end tr_TBL1_After_Modify;
```

После осуществления операций добавления (insert), обновления (update) и удаления (delete) данных в таблице TBL_1, в таблицу TBL_1_Event добавляются соответствующие записи. Но теперь нам нужно сделать так, чтобы подобный триггер создавался автоматически при создании рабочей таблицы. Для этого необходимо при создании таблицы-клона (вызове процедуры CreateCloneTable) реализовать операцию создания подобного триггера.

Добавим в тело пакета несколько функций. Одной из таких функций будет ListFields (Листинг 10), которая принимает на вход строку, содержащую имя таблицы, и возвращает строку, содержащую все поля (кроме ID или OID — возможное название первичных ключей), перечисленные через запятую.

Листинг 10. Функция ListFields, возвращающая список полей указанной в аргументе таблицы.

```
function ListFields (tablename in varchar2, prefix in varchar2)
  return varchar2
IS
  result varchar2(1000);
BEGIN
  for rec in (select C.COLUMN_NAME,
                     sysdate,
                     0,
                     C.TABLE_NAME,
                     C.DATA_TYPE,
                     C.CHAR_LENGTH,
                     C.NULLABLE
                from ALL_TAB_COLUMNS C
                where UPPER(C.OWNER) = UPPER(SYS_CONTEXT('USERENV',
                                    'CURRENT_SCHEMA'))
                     and trim(C.TABLE_NAME) = trim(tablename))
  loop
    if (rec.COLUMN_NAME<>'OID' and rec.COLUMN_NAME<>'ID') then
      result := result || prefix || rec.COLUMN_NAME || ',';
    end if;
  end loop;

  return result;
end;
```

Функцией ListFields мы воспользуемся при формировании сценария, создающего триггер для фиксирования всех изменений в создаваемой таблице. Функция CreateTriggerModify (Листинг 11) возвращает строку со сценарием создания триггера. Из кода видно, что мы считаем OID первичным ключом в рабочей таблице.

Листинг 11. Функция CreateTriggerModify возвращает сценарий создания триггера, фиксирующего все изменения данных рабочей таблицы в таблице-клоне.

```
-- «Создает» (возвращает скрипт для создания) триггера на модификацию
-- данных в рабочей таблице (INSERT, UPDATE, DELETE)*/
function CreateTriggerModify (tablename in varchar2, tablenameClone in varchar2)
          return varchar2
is
  result varchar2(10000);
begin
  result := ' create or replace trigger «' || cntPrefixTrigger || tablename
           || '_Modify» after insert or update or delete on «'
           || tablename || '»
  for each row
  declare
  begin
```

```
  if (inserting) then
   insert into ' || '»' || tablenameClone
           || '» (';
   result := result || ListFields(tablename, '');
   result := result || '»RecId», «Operation») VALUES(';
   result := result || ListFields(tablename, ':new.');
   result := result || ' :new.oid, 0);
  end if;
  if (updating) then
   insert into ' || '»' || tablenameClone || '» (';
   result := result || ListFields(tablename,'');
   result := result || '»RecId», «Operation») VALUES(';
   result := result || ListFields(tablename,':new.');
   result := result ||' :new.oid, 1);
  end if;
  if (deleting) then
   insert into ' || '»' || tablenameClone || '» (';
   result := result || ListFields(tablename,'');
   result := result || '»RecId», «Operation») VALUES(';
   result := result || ListFields(tablename,':old.');
   result := result ||' :old.oid, 2);
  end if;
  end «' || cntPrefixTrigger || tablename || '_Modify»;';

 return result;
end;
```

Теперь необходимо внести небольшие дополнения в процедуру CreateCloneTable. В первую очередь добавим переменную:

```
sqlCreateScriptTriggerModify varchar2(5000).
```

Затем обратимся к функции CreateTriggerModify, передав ей имя рабочей таблицы и имя таблицы-клона:

```
sqlCreateScriptTriggerModify := CreateTriggerModify(tableName, tableNameClone);
```

Дополним процесс создания задания (job) строкой с новым сценарием создания триггера:

```
dbms_job.submit(job => job_no_out, what =>
  'begin ' || chr(10)
  || CreateScriptForImmedate(
      CreateScriptForBegin(sqlCreateScriptSequence || chr(10)
        || sqlCreateScriptTable || chr(10)
        || sqlCreateScriptStructureTable || chr(10)
        || sqlCreateScriptTrigger
        || chr(10) || sqlCreateScriptTriggerModify)
    )
  || chr(10) || 'end;',
```

Предположим, что после всех наших манипуляций мы создали рабочую таблицу со структурой: OID, LNAME, MNAME, FNAME. Автоматически будет создана таблица-клон по ранее описанным правилам. При добавлении, изменении и удалении данных рабочей таблицы все изменения будут зафиксированы в таблице-клоне (рисунок 6).

	OID	LNAME	FNAME	MNAME	UserName	RecId	DateChange	Operation
1	1	И1_1	И1_2	И1_3	METAXPO	1	18.02.2010 16:59:00	0
2	2	И2_1	И2_2	И2_3	METAXPO	2	18.02.2010 16:59:00	0
3	3	И1_1v	И1_2	И1_3	METAXPO	1	18.02.2010 16:59:11	1
4	4	И2_1c	И2_2	И2_3	METAXPO	2	18.02.2010 16:59:11	1
5	5	И1_1v	И1_2	И1_3	METAXPO	1	18.02.2010 16:59:16	2

Рисунок 6. Содержимое таблицы-клона после осуществления манипуляций с данными в рабочей таблице.

Посмотрев на рисунок, вы сможете без труда сказать, что происходило, когда и кем осуществлялись манипуляции с данными в рабочей таблице.

Заключение

Мы описали подход, который использовали для того, чтобы вести аудит изменений в схеме данных СУБД Oracle. Данный подход с вариациями может применяться в различных СУБД, которые поддерживают DDL-триггеры, т.е. триггеры, реагирующие на изменение структуры схемы данных. Подробно не была освещена реакция триггера при модификации рабочих таблиц, т.е. команды ALTER, этот вопрос можно было бы осветить во второй части статьи. Логическим продолжением данной статьи может являться исследование ведения аудита с клиента, а не СУБД, что позволило бы реализовывать более сложную логику и снизить нагрузку на сервер Oracle.

Список использованной литературы

1. Урман, С. Oracle8: Программирование на языке PL/SQL. — М.: Изд-во Лори, 1999. — 607 с.
2. Кривонос, Н. Журналирование изменений структуры БД и данных. — URL: http://www.compdoc.ru/bd/sql/log_change_of_structure_bd.
3. Finningan, P. Introduction to Simple Oracle Auditing. — URL: http://www.securityfocus.com/infocus/1689.
4. Кайт, Т. Эффективное проектирование приложений Oracle. — М.: Изд-во «Лори», 2006. — 637 с.
5. Oracle 9i Application Developer's Guide — Fundamentals. — URL: http://download-west.oracle.com/docs/cd/B105-01_01/appdev.920/a96590/adg14evt.htm.
6. Кайт, Т. Триггеры на операторы ЯОД и аудит изменений структуры базы — URL: http://www.ln.com.ua/~openxs/projects/oracle/ora022.html).

❖