Transactions

9 min. read · View original

In the introduction we covered some examples of inserting data into a database by typing at SQL*Plus:

```
insert into mailing_list (name, email)
values ('Philip Greenspun','philg@mit.edu');
```

Generally, this is not how it is done. As a programmer, you write code that gets executed every time a user submits a discussion forum posting or classified ad. The structure of the SQL statement remains fixed but not the string literals after the `values`.

The simplest and most direct interface to a relational database involves a procedural program in C, Java, Lisp, Perl, or Tcl putting together a string of SQL that is then sent to to the RDBMS. Here's how the ArsDigita Community System constructs a new entry in the clickthrough log:

```
insert into clickthrough_log
 (local_url, foreign_url, entry_date,
click_count)
values
 ('$local_url', '$foreign_url', trunc(sysdate),
1)"
```

The INSERT statement adds one row, filling in the four list columns. Two of the values come from local variables set within the Web server, `$local_url` and `$foreign_url`. Because these are strings, they must be surrounded by single quotes.

One of the values is dynamic and comes straight from Oracle: `trunc(sysdate)`. Recall that the `date` data type in Oracle is precise to the second. We only want one of these rows per day of the year and hence truncate the date to midnight. Finally, as this is the first clickthrough of the day, we insert a constant value of 1 for `click_count`.

Atomicity

Each SQL statement executes as an atomic transaction. For example, suppose that you were to attempt to purge some old data with

```
delete from clickthrough_log where entry_date +
120 < sysdate;
```

(delete clickthrough records more than 120 days old) and that 3500 rows in `clickthrough_log` are older than 120 days. If your computer failed halfway through the execution of this DELETE, i.e., before the transaction committed, you would find that none of the rows had been deleted. Either all 3500 rows will disappear or none will.

More interestingly, you can wrap a transaction around multiple SQL statements. For example, when a user is editing a comment, the ArsDigita Community System keeps a record of what was there before:

```
ns_db dml $db "begin transaction"

# insert into the audit table
ns_db dml $db "insert into general_comments_audit
 (comment_id, user_id, ip_address,
audit_entry_time, modified_date, content)
select comment_id,
       user_id,
       '[ns_conn peeraddr]',
       sysdate,
       modified_date,
       content from general_comments
where comment_id = $comment_id"
```

```
# change the publicly viewable table
ns_db dml $db "update general_comments
set content = '$QQcontent',
    html_p = '$html_p'
where comment_id = $comment_id"

# commit the transaction
ns_db dml $db "end transaction"
```

This is generally referred to in the database industry as *auditing*. The database itself is used to keep track of what has been changed and by whom.

Let's look at these sections piece by piece. We're looking at a Tcl program calling AOLserver API procedures when it wants to talk to Oracle. We've configured the system to reverse the normal Oracle world order in which everything is within a transaction unless otherwise committed. The `begin transaction` and `end transaction` statements never get through to Oracle; they are merely instructions to our Oracle driver to flip Oracle out and then back into autocommit mode.

The transaction wrapper is imposed around two SQL statements. The first statement inserts a row into `general_comments_audit`. We could simply query the `general_comments` table from Tcl and then use the returned data to create a standard-looking INSERT. However, if what you're actually doing is moving data from one place within the RDBMS to another, it is extremely bad taste to drag it all the way out to an application program and then stuff it back in. Much better to use the "INSERT ... SELECT" form.

Note that two of the columns we're querying from `general_comments` don't exist in the table: `sysdate` and `'[ns_conn peeraddr]'`. It is legal in SQL to put function calls or constants in your select list, just as you saw at the beginning of the Queries chapter where we discussed Oracle's one-row system table: `dual`. To refresh your memory:

```
select sysdate from dual;

SYSDATE
----------
1999-01-14
```

You can compute multiple values in a single query:

```
select sysdate, 2+2, atan2(0, -1) from dual;

SYSDATE           2+2 ATAN2(0,-1)
---------- ---------- -----------
1999-01-14          4  3.14159265
```

This approach is useful in the transaction above, where we combine information from a table with constants and function calls. Here's a simpler example:

```
select posting_time, 2+2
from bboard
where msg_id = '000KWj';

POSTING_TI        2+2
---------- ----------
```

```
1998-12-13          4
```

Let's get back to our comment editing transaction and look at the basic structure:

- open a transaction
- insert into an audit table whatever comes back from a SELECT statement on the comment table
- update the comment table
- close the transaction

Suppose that something goes wrong during the INSERT. The tablespace in which the audit table resides is full and it isn't possible to add a row. Putting the INSERT and UPDATE in the same RDBMS transactions ensures that if there is a problem with one, the other won't be applied to the database.

Consistency

Suppose that we've looked at a message on the bulletin board and decide that its content is so offensive we wish to delete the user from our system:

```
select user_id from bboard where msg_id =
'000KWj';

   USER_ID
----------
     39685

delete from users where user_id = 39685;
*
ERROR at line 1:
ORA-02292: integrity constraint
(PHOTONET.SYS_C001526) violated - child record
found
```

Oracle has stopped us from deleting user 39685 because to do so would leave the database in an inconsistent state. Here's the definition of the bboard table:

```
create table bboard (
        msg_id          char(6) not null primary
key,
        refers_to       char(6),
        ...
        user_id         integer not null
references users,
        one_line        varchar(700),
        message         clob,
        ...
);
```

The `user_id` column is constrained to be not null. Furthermore, the value in this column must correspond to some row in the `users` table (`references users`). By asking Oracle to delete the author of msg_id `000KWj` from the `users` table before we deleted all of his or her postings from the `bboard` table, we were asking Oracle to leave the RDBMS in an inconsistent state.

Mutual Exclusion

When you have multiple simultaneously executing copies of the same program, you have to think about *mutual exclusion*. If a program has to

- read a value from the database
- perform a computation based on that value
- update the value in the database based on the computation

Then you want to make sure only one copy of the program is executing at a time through this segment.

The /bboard module of the ArsDigita Community System has to do this. The sequence is

- read the last message ID from the `msg_id_generator` table

- increment the message ID with a bizarre collection of Tcl scripts
- update the `last_msg_id` column in the `msg_id_generator` table

First, anything having to do with locks only makes sense when the three operations are grouped together in a transaction. Second, to avoid deadlocks a transaction must acquire all the resources (including locks) that it needs at the start of the transaction. A SELECT in Oracle does not acquire any locks but a SELECT .. FOR UPDATE does. Here's the beginning of the transaction that inserts a message into the `bboard` table (from /bboard/insert-msg.tcl):

```
select last_msg_id
from msg_id_generator
for update of last_msg_id
```

Mutual Exclusion (the Big Hammer)

The `for update` clause isn't a panacea. For example, in the Action Network (described in Chapter 16 of Philip and Alex's Guide to Web Publishing), we need to make sure that a double-clicking user doesn't generate duplicate FAXes to politicians. The test to see if the user has already responded is

```
select count(*) from an_alert_log
where member_id = $member_id
and entry_type = 'sent_response'
and alert_id = $alert_id
```

By default, Oracle locks one row at a time and doesn't want you to throw a FOR UPDATE clause into a SELECT COUNT(*). The implication of that would be Oracle recording locks on every row in the table. Much more efficient is simply to start the transaction with

```
lock table an_alert_log in exclusive mode
```

This is a big hammer and you don't want to hold a table lock for more than an instant. So the structure of a page that gets a table lock should be

- open a transaction

- lock table
- select count(*)
- if the count was 0, insert a row to record the fact that the user has responded
- commit the transaction (releases the table lock)
- proceed with the rest of the script
- ...

What if I just want some unique numbers?

Does it really have to be this hard? What if you just want some unique integers, each of which will be used as a primary key? Consider a table to hold news items for a Web site:

```
create table news (
        title           varchar(100) not null,
        body            varchar(4000) not null,
        release_date    date not null,
        ...
);
```

You might think you could use the `title` column as a key, but consider the following articles:

```
insert into news (title, body, release_date)
values
('French Air Traffic Controllers Strike',
 'A walkout today by controllers left travelers
stranded..',
 '1995-12-14');

insert into news (title, body, release_date)
values
('French Air Traffic Controllers Strike',
 'Passengers at Orly faced 400 canceled flights
...',
 '1997-05-01');

insert into news (title, body, release_date)
values
('Bill Clinton Beats the Rap',
 'Only 55 senators were convinced that President
Clinton obstructed justice ...',
 '1999-02-12');

insert into news (title, body, release_date)
values
('Bill Clinton Beats the Rap',
 'The sexual harassment suit by Paula Jones was
dismissed ...',
 '1998-12-02);
```

It would seem that, at least as far as headlines are concerned, little of what is reported is truly new. Could we add

```
        primary key (title, release_date)
```

at the end of our table definition? Absolutely. But keying by title and date would result in some unwieldy URLs for editing or approving news articles. If your site allows public suggestions, you might find submissions from multiple users colliding. If you accept comments on news articles, a standard feature of the ArsDigita Community System, each comment must reference a news article. You'd have to be sure to update both the comments table and the news table if you needed to correct a typo in the `title` column or changed the `release_date`.

The traditional database design that gets around all of these problems is the use of a generated key. If you've been annoyed by having to carry around your student ID at MIT or your patient ID at a hospital, now you understand the reason why: the programmers are using generated keys and making their lives a bit easier by exposing this part of their software's innards.

Here's how the news module of the ArsDigita Community System works, in an excerpt from http://software.arsdigita.com/www/doc/sql/news.sql:

```

create sequence news_id_sequence start with 1;
create table news (
        news_id         integer primary key,
        title           varchar(100) not null,
        body            varchar(4000) not null,
        release_date    date not null,
        ...
);
```

We're taking advantage of the nonstandard but very useful Oracle *sequence* facility. In almost any Oracle SQL statement, you can ask for a sequence's current value or next value.

```
SQL> create sequence foo_sequence;

Sequence created.

SQL> select foo_sequence.currval from dual;

ERROR at line 1:
ORA-08002: sequence FOO_SEQUENCE.CURRVAL is not
yet defined in this session
```

Oops! Looks like we can't ask for the current value until we've asked for at least one key in our current session with Oracle.

```
SQL> select foo_sequence.nextval from dual;

   NEXTVAL
----------
         1

SQL> select foo_sequence.nextval from dual;

   NEXTVAL
----------
         2

SQL> select foo_sequence.nextval from dual;

   NEXTVAL
----------
```

```
         3

SQL> select foo_sequence.currval from dual;

   CURRVAL
----------
         3
```

You can use the sequence generator directly in an insert, e.g.,

```

insert into news (news_id, title, body,
release_date)
values
(news_id_sequence.nextval,
 'Tuition Refund at MIT',
 'Administrators were shocked and horrified ...',
 '1998-03-12);
```

Background on this story:
http://philip.greenspun.com/school/tuition-free-mit.html

In the ArsDigita Community System implementation, the `news_id` is actually generated in /news/post-new-2.tcl:

```

set news_id [database_to_tcl_string $db "select
news_id_sequence.nextval from dual"]
```

This way the page that actually does the database insert, /news/post-new-3.tcl, can be sure when the user has inadvertently hit submit twice:

```

if [catch { ns_db dml $db "insert into news
(news_id, title, body, html_p, approved_p,
 release_date, expiration_date, creation_date,
creation_user,
 creation_ip_address)
values
($news_id, '$QQtitle', '$QQbody', '$html_p',
'$approved_p',
 '$release_date', '$expiration_date', sysdate,
$user_id,
 '$creation_ip_address')" } errmsg] {
    # insert failed; let's see if it was because
of duplicate submission
    if { [database_to_tcl_string $db "select
count(*)
                                      from news
                                      where
news_id = $news_id"] == 0 } {
        # some error other than dupe submission
        ad_return_error "Insert Failed" "The
database ..."
        return
    }
    # we don't bother to handle the cases where
there is a dupe submission
    # because the user should be thanked or
redirected anyway
}
```

In our experience, the standard technique of generating the key at the same time as the insert leads to a lot of duplicate information in the database.

Sequence Caveats

Oracle sequences are optimized for speed. Hence they offer the minimum guarantees that Oracle thinks are required for primary key generation and no more.

If you ask for a few nextvals and roll back your transaction, the sequence will not be rolled back.

You can't rely on sequence values to be, uh, sequential. They will be unique. They will be monotonically increasing. But there might be gaps. The gaps arise because Oracle pulls, by default, 20 sequence values into memory and records those values as used on disk. This makes nextval very fast since the new value need only be marked use in RAM and not on disk. But suppose that someone pulls the plug on your database server after only two sequence values have been handed out. If your database administrator and system administrator are working well together, the computer will come back to life running Oracle. But there will be a gap of 18 values in the sequence (e.g., from 2023 to 2041). That's because Oracle recorded 20 values used on disk and only handed out 2.

More

Next: Triggers

---

philg@mit.edu

Add a comment