STATEFULL VS STATELESS FOR DATABASE DEVELOPERS

Michael Rosenblum, Dulcian, Inc.

Introduction

Unless you attended college earlier than 1995, the majority of your early experience and training was based on Client-Server architecture. One of the core parts of that architecture is the concept of a "session," namely a set of activities between logon and logoff. When large web-based IT systems became necessary, developers and architects started to apply existing techniques because they worked. Major software companies followed that lead, the best example of which is Oracle Forms 6*i* which was an attempt to directly port a perfectly designed and extremely popular environment to the new web world. At first glance, the learning curve appeared simple, but after the first few deployments, this whole universe started to collapse. A critical missing part of the original analysis was the increase in users by orders of magnitude. The rules applicable to 100 connections just didn't work for 100,000 connections. The solution seemed simple: introduce a separation between logical and physical sessions.

- Physical session – a set of activities in the context of one server connection. Depending upon the implementation technology, there could be two different approaches:
  - Full cycle – A session that consists of request→processing→response as a complete set starting from the moment the request is initiated up to the point when the last part of the response is interpreted.
  - One-way – Two completely different queues (request and response), where both events can occur independent of each other. Requests are sent without waiting and a special listener retrieves responses as soon as they are ready.
- Logical session – a set of activities between user logon and logoff that consists of a number of physical sessions. Each physical session is completely independent from the next/previous one. In other words, there is no such thing as the real state of the session. It is the developer's responsibility to capture enough information to simulate the persistence of a logical session. That architecture is called *stateLESS* to differentiate it from the old *stateFULL* architecture where physical session was always equal to logical.

Web applications must be stateless, otherwise they will not scale. We recently heard about one existing Forms-based solution that required 16 racks of servers to support about 2000 simultaneous users. So, the real question is: Where to store persistent data? The choices are fairly obvious: in the client, in the middle tier or in the database. Since Dulcian uses a database-centric approach, this paper will discuss the problem of stateFULL vs. stateLESS from that perspective. Our target audience is database developers and DBAs who need to support the architectural decisions of their front-end teams or influence these decisions one way or another.

I. You can't leave everything to the Oracle DBMS any more

It is very difficult to change already established development patterns. However, in order to be able to step up to the world of web-based solutions, you will probably need to forget a lot of old patterns, or at least change your definitions, It is common knowledge that:

- Oracle handles locks better than anything you can develop.
- Packaged variables are the best way to store and quickly access data.
- Temporary tables are one of the most useful techniques to cache intermittent information
- Savepoints can be extremely useful, especially for bulk processing.
- Continuing from a database error is easy.

This list could be made very long, but the main idea is that there is a habit of trusting the Oracle DBMS engine to do everything possible. This approach is correct up to a point, but in the StateLESS environment there is no persistent DBMS engine behind you. You have to start reinventing the wheel.

A. Locking

One of the most popular techniques for client/server development was using the `SELECT…FOR UPDATE NOWAIT` structure. Using this approach, either the code works and you can proceed, or it immediately fails, which means that somebody else is working with the record you are trying to access. Another similar technique involved firing an early UPDATE on the record in order to lock it for future

updates (even an initial update was completely unnecessary). In the stateless world, since there is no persistent database connection, neither of these techniques helps because all Oracle locks involve session-level resources.

In the stateless environment, the only approach is to use a unique session identifier (usually cached as a "cookie" on the client side). This means that locking the record is the same as somehow associating that session ID with the primary key of the record you are trying to lock. But you just cannot add an extra column to every table. At Dulcian, we have found that the best approach is to group data hierarchically and implement check-in/check-out mechanisms at the root level.

For example, assume that you have a customer with a lot of personal information (phone, address, relatives, school, etc). An extra column is added only to the CUSTOMER table. In order to change any piece of information you must have that customer's file under your full control. Even supervisors should not be able to make any modifications until you release the lock. Using this approach, the set of corresponding APIs consists of the following:

- Check-in – write session ID to the CUSTOMER table
- Check-out – remove session ID from the CUSTOMER table
- Can Access (passing session ID) – returns *true* if session ID in the CUSTOMER table is yours or NULL, otherwise return *false*
- Unlock (passing user ID) – since web-connections are not stable, it is possible to lose your session ID cached in your browser. The best way to handle this is to check for any previous sessions not correctly closed any time the user logs in, and reset all checked-out objects to the safe point of the processing flow.
- Administrative Unlock – special help-desk module to get a customer out of a locked state in case there is no time to wait for the processing person to re-login (e.g. somebody left for vacation and didn't close all sessions)

By implementing these APIs, the whole development cycle became much easier since it was very clear how to check access rights and identify the points where data ownership is transferred.

Keep the following rules in mind:

- To implement locking in the stateless environment, add the session ID to the appropriate number of tables.
- These tables should be grouped hierarchically to simplify application logic and decrease the number of checks made each time a user needs to change a piece of data.

B. Package Variables and Temporary Tables

The last section discussed the most widely used session-level resources, but there are a number of other database elements for which the lifecycle is defined between database logon and logoff. From a developer's point of view, the most crucial of these are package variables and temporary tables. Both of these concepts serve exactly the same purpose, namely caching information in the middle of the processing cycle for the sake of performance optimization. There is no need to recalculate what is already known.

Technically, using package variables is fine inside of the physical session. However, this implies that there should be a special mechanism to load/unload them between physical sessions. Doing this is fairly simple by creating a table where each row represents one session. In that table, there should either be a set of columns (one for each variable) or a single CLOB and a set of the following APIs:

- Initialize Globals – fired at the moment of user login; creates a new row with initial values for each registered package variable (These could either be hard coded or stored in the special repository and retrieved using dynamic SQL.)
- Write Globals – take packaged variables and update their values with a SESSION INFO table
- Read Globals – read the SESSION INFO table and set appropriate values for all required variables
- Cleanup info-table – since client sessions could be dropped, there is always a chance of having "leftovers" in the SESSION INFO table. Therefore, a daily cleanup routine is very useful. Usually, removing sessions that are more than two days old is sufficient for systems where the expiration time is set to 8 hours of inactivity.

Temporary tables are a different story. They are usually used as intermediate storage points for large calculation routines, but sometimes they are also read directly (OLTP DML operations are very rare).

The logic should involve just two steps: preparation of the data and retrieval of it afterwards. For these situations, there is a direct substitution. Assume the existence of the following in a client/server environment:

- Temporary table *T1(a number, …, z number)*
- Routine to populate the temporary table *P1(in_dt date, in_dept number)* – normally this kind of logic is used when there is no direct way of using plain SQL to transform stored data into the required data representation.
- View to present results *V1(a1, z1) as select … from T1*

In the stateless world these structures would be changed to the following:

- Object type *T1_OT (a1, … z1)* – the same structure as view V1. Since you are hiding the processing logic in the PL/SQL code, the only thing that matters is a final output.
- Object collection *T1_TT is table of T1_OT*
- Function that takes the same input as P1 and returns T1_TT: *f1(in_dt date, in_dept number) return T1_TT*
- Prepared statement with appropriate bind variables in the middle tier: *select … from table (cast (f1(:1,:1) as T1_TT))* – since it is impossible to pass parameters to a regular Oracle view, the best way is to use a simple query, even if this slightly violates the purity of the database-centric approach.

Life becomes much more difficult if your temporary tables were used for any multi-step processing or DML from the front-end. In the case of multi-step processing, the only way out is to develop your system in such a way that all required steps occur inside of the same physical session. For example, complete verification of personnel records could be made into a single request as follows:

- Extract required data into TEMP table – if any errors occur, stop and generate a report
- Standardize extracted data – if any errors occur, stop and generate a report
- Validate standardized data against a set of business rules – if any errors occur, stop and generate a report
- Generate a confirmation unless any errors occurred previously

Using this approach, although you lose the possibility of seeing the intermediate steps of verification, you can be 100% sure that data will never be corrupted.

If DML is required, there is no other way around. You must have a real table with a session ID as one of its columns. This approach should be used with caution since you must have the session ID in every query to access that table. Doing it this way is very resource-intensive, but there are ways around it that will be covered in later in this paper.

Keep the following rules in mind when dealing with package variables and temporary tables in a stateless environment:

1. Packaged variables are stored as one set per session
2. Single-cycle usage of temporary tables can be converted into functions that return object collections.
3. Temporary tables that are accessed multiple times must become persistent and structured by session ID.

C. Error handling

In a normal Client-Server environment, an error could mean any failure at the database level, such as primary key violations, check constraint violations, etc. These errors were either handled by the software or propagated up to the client side with some kind of a user-friendly message. To be fair, in every Client-Server system with which this author has ever worked, there were at least a few dozen *raise_application_error(-20001, 'You cannot do it!')* of some kind. The critical point here is that after the error was displayed/handled, you could still continue with appropriate actions.

In the stateless environment, an error involves a failure of the entire request. Since every physical session is a completely new "creature," the safest thing to do if something goes wrong is to failover to the state of data that existed at the moment the initial request is received. Since our world is highly modularized, the easiest way to fill half-full teapot is to use a routine like "empty teapot" followed by a routine "fill teapot" rather than try to create a new routine "add what's required". Although this may be a waste of already processed information, there are too many things that may go wrong (e.g. external services, multi-layer data storages, Internet connectivity issues, etc). Since the total development time required to handle anything that can possibly occur is usually beyond any reasonable budget, there should be no savepoints, no intermediate commits, and there should always be a simple way to restore the initial state of the data.

It is important to distinguish an error from an Oracle exception. They are not the same. In Oracle Forms, it was very easy to show an Oracle error in a less alarming way, but not all environments are so friendly. In many contemporary development environments, a

basic Oracle error generates a half-page of stack trace on the Java side. Also, if you are using Oracle's Application Development Framework – Business Components (ADF BC) or any other similar mechanism using the "dirty flag" concept, a real database error could create havoc in the cached dataset. A good rule of thumb to follow is that, under no circumstances should a database error reach even the middle tier. There are two major ways to implementing this:

1. Error queue – the best option when requests are very granular
    - After every request, a special call is made to identify any problems with the current session ID in the error queue.
    - All routines have an exception block that writes enough information to the error queue so that the program can be restored to a stable state. An appropriate message is displayed (if needed), and required actions are taken.
2. Request wrapper – single point of contact for all kinds of requests
    - Special function that takes needed input (maybe even as a CLOB)
    - Function returns a string with one of three possible types of feedback:
        - Error message – since the whole request is wrapped, it could self-restore the data state so the only task left is to notify the user
        - Question/Warning – if the request required additional information, available options with a question could be returned with some hints about formatting/display data, etc.
        - Confirmation message

As a result of either of these implementations, both the front-end and middle tier recognize that anything coming from the database is 100% predictable. It is always easier to tweak something directly in the area where the code is generated rather than in a different layer. Keep the following rules in mind:

1. The best way out is to failover to the very beginning
2. Oracle errors should never leave the database

II Connection Pooling

Since, in the stateless environment, there is no need to have persistent sessions, the total number of physical sessions at any point in time is fairly small. If 1000 people click every 10 seconds and each request takes 0.2 sec to process, the average workload on the system will only be 20 sessions. Rather than opening/closing a physical session every time a new request comes in to an application you can create a connection pool with a fixed number of connections (with autoextend option) and serve them to incoming requests as needed.

This brilliant idea caused huge problems for anyone who tried to implement it for the first time. Now you have the opposite problem, namely a single physical session that can serve requests from different logical sessions at different points in time. This adds new meaning to the term "dirty data" since you cannot trust ANYTHING defined at the session level. In this case, packaged variables can be handled with a few lines of code. Both procedures take very little time to fire:

```
begin
dbms_session.reset_package; -- reset all packaged variables to the initial state
dbms_session.free_unused_user_memory; -- release all memory freed by previous state
end;
```

But the question of temporary tables is more difficult to resolve. There is no simple way to identify what tables have data, or clean that data. The following routine could be somewhat useful (even if it is somewhat overkill):

```
procedure p_truncate
is
    v_exist_yn varchar2(1);
begin
    select 'Y'
    into v_exist_yn
    from v$session s,
         v$tempseg_usage u
    where s.audsid = SYS_CONTEXT('USERENV','SESSIONID')
    and   s.saddr = u.session_addr
    and   u.segtype = 'DATA'
    and rownum = 1;
```

```
    for c in (select table_name
              from user_tables
              where temporary = 'Y'
              and duration = 'SYS$SESSION')
    loop
        execute immediate 'truncate table '||c.table_name;
    end loop;
end;
```

The idea is simple. Since using *V$TEMPSEG_USAGE* makes it possible to detect whether or not the current session has temporary segments allocated, the cycle of cleanups can be avoided in most cases. However, the Oracle DBMS does not release the TEMP tablespace allocated to temporary CLOBs (all CLOB variables) until the end of a session. This is the expected behavior, which is why segment type should be filtered. According to Metalink ID 5723140 in 10.2.0.4 and 11.1.0.6, Oracle introduced event 60025 to get around the described behavior, but caution is strongly recommended.

III. Resource Utilization

In the case of a statefull environment with a temporary table containing a lot of DML, assume, that there is a special engine that keeps a lot of supporting information about the application (literally, hundreds of rows) in the temporary table. The application always reads from that support area. To write, the application first tells the engine what should be done, and the engine update support area and application read the results. This mechanism eliminates about 75% of all unnecessary repeated requests and makes a system much more robust.

Now there is a need to go stateless. The basic solution was just to add session ID as a separate column. Because of a lot of transactional activities, original estimates indicated that the system could slow down 3-5%. In real life, the situation was much worse since everything became 50%-200% slower, but only at peak times. There appeared to be some kind of a workload limit after which the whole system started to fall apart.

Additional research efforts revealed the following:

- The database was running in ARCHIVELOG so all DML against that support table was recorded and filled up about 85% of all logs (proved by LogMiner).
- Since all of these support changes must be persistent, a lot of extra COMMITS occurred and, as a result, the LOG FILE SYNC wait event count skyrocketed.
- The table had a primary key (synthetic ID from a sequence), but because there was a lot of DML activity from hundreds of sessions, about every 15 minutes the database was logging a deadlock in addition to very high contention on some index blocks.
- Because of cumulative heavy I/O load, individual requests started to take more time. As a result, physical sessions were not being released from the connection pool fast enough and the total number of simultaneous sessions increased about 4 times more than originally estimated.
- Since the total number of sessions increased, each session used more memory, more temporary segments, etc. which slowed down the system even more. This was especially true for I/O operations (since there were more simultaneous requests). This situation quickly spirals into a slow-down and eventual stoppage of the system.

In this example, every possible database resource quickly became over-utilized just by making a table persistent with a session key. The two core issues were how to decrease I/O and how to resolve the index contention. There was a workaround using the following techniques:

- Adding extra memory to the server permitted the creation of a separate database instance
    - The new instance runs in NOARCHIVELOG mode
    - The new instance has only one schema.
    - That schema contains only one table : SUPPORT INFO
    - The SUPPORT INFO table is hash-partitioned by session ID (1024 partitions)
    - All indexes are local.
- The main schema has a database link and a synonym so everything appears as though nothing has changed

  - All requests to the support table must include a session ID (to use local indexes). Some rewrite was required to enforce this rule.

As a result of the changes listed above, as the system ran as fast as originally predicted (extra waits, caused by data cases via DBLink, were negligible, less than 0.01/request with an average of 3000 requests/hour). The results were as follows:

- Because no time was lost writing logs, there was less IO → less session → less resources used → less waits → faster response → less sessions …
- Using a large number of partitions, there was less chances of creating a "hot block", especially since all indexes were local.

The most critical lesson learned from this use case was that, in the Oracle environment, everything is linked together. Therefore, any changes can lead to a "domino effect," especially when working in an unfamiliar, i.e. stateless environment.

Conclusions

There is no way to build any reasonable web-based solution without going stateless. The questions are: at what point, on what level and how much effort does this take? There are not a lot of options for handling this situation correctly but there are many hidden traps. It is impossible to cover all of these in a single paper, but the purpose was to make readers aware of the potential pitfalls and areas where caution is critical.

About the Author

Michael Rosenblum is a Development DBA at Dulcian, Inc. He is responsible for system tuning and application architecture. He supports Dulcian developers by writing complex PL/SQL routines and researching new features. Mr. Rosenblum is the co-author of *PL/SQL for Dummies* (Wiley Press, 2006). Michael is a frequent presenter at various regional and national Oracle user group conferences. In his native Ukraine, he received the scholarship of the President of Ukraine, a Masters Degree in Information Systems, and a Diploma with Honors from the Kiev National University of Economics.