IMPLEMENTING CONNECTION POOLS FOR DATA-CENTRIC APPLICATIONS

Michael Rosenblum, Dulcian, Inc.

Introduction

When one is given two appealing options, it is human nature to try to get the best of both. However, there are always advantages and disadvantages to most alternatives. The story behind connection pools is no different.

In the mid 1990's, developers and DBAs discovered that keeping persistent database sessions for every client connection is technically impossible when building scalable web-based IT solutions. Developers needed additional tools and strategies for these types of systems. To satisfy this need, system architects introduced the concept of separating logical and physical database sessions. The following descriptions explain the differences between the two:

- Physical session – a set of activities in the context of one server connection. Depending upon the implementation technology, there are two different approaches:
    - Full cycle – A session that consists of request→processing→response as a complete set starting from the moment that 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 independently. 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. This architecture is called *stateLESS* to differentiate it from the old *stateful* architecture where one physical session was always equal to one logical session.

Stateful and Stateless System Architectures

Both stateful and stateless architectures each have their own advantages and disadvantages. A few are listed here:.

Stateful

Stateful systems have the following advantages:

- They include a predictable and reasonable number of connections.
- A predictable number of resources are required to keep system running.
- There is the possibility of using session-level features to optimize performance (temporary tables, packaged variables, etc.)
- Since all activities are happening in the same session, there is no need to reload packages/execution plans to the memory.

The main (but significant disadvantage is that stateful systems do not scale well

Stateless

Stateless systems have the following advantages:

- At any point in time, there are only a small number of sessions connected to the database
- The workload typically follows a statistical trend.

The following disadvantages can be associated with stateless system architectures:

- Keeping a persistent layer is difficult and there are different schools of thought about where to place it (database/middle tier/client), not to mention how to go about it. (For more details about this topic, see my RMOUG 2008 paper “Stateful vs. StateLESS for Database Developers”).
- Each physical session must be opened and closed. If you do this thousands of times, it becomes expensive, especially if your code is PL/SQL-intensive, because each package must be reloaded and reinitialized
- Workload is more or less statistically even, so it is difficult to manage possible unpredicted activity spikes.

It is clear, that stateless solutions were able to solve the core scalability problem since they made it possible to build systems that would scale up to thousands if not hundreds of thousands of simultaneous users. However, the costs were too high for the following reasons:

1. Managing the persistent layer is time-intensive.
2. There are significant performance impacts of the activities required to manage a huge number of separate physical requests.
3. There is only a low level of control over how many sessions are executed at any point in time.

Connection Pools

The result of trying to get the best of both stateful and stateless architectures was the concept of *connection pools*. From the big picture point of view, everything looked simple:

- The middle tier creates a small set of physical connections to the database.
- When an incoming request comes to the middle tier, it serves the next free session from the pool to the request (instead of opening a new session for each request).
- If all sessions are busy, the middle tier adds extra ones to the connection pool.

However, the actual implementation of connection pools is significantly more challenging and includes very different types of problems: pool management, training issues, session resource management, and database resource management.

Managing Connection Pools

Connection pools must have a number of core managing elements:

1. Upper bound of connections in the connection pool
   - It is recommended that you have a delay option, when a request could wait for some time until a free session from the pool is found. Since users of web applications are accustomed to network glitches of some kind, they will not be surprised by an extra few seconds of wait time, but the cost of a failed request could be too high. It is almost impossible to code an application where you assume that any request could break. As a result, the recovery process may require a lot of manual effort.
   - Each failed request should be logged. If the system hits an upper bound, it is either set incorrectly, or something is very wrong.
2. Randomization of connection assignments from the pool may be a good or bad idea, depending upon the actual situation.
   - If you do not randomize connections (done in a majority of implementations), the number of sessions at any point of time is very small, but the workload of these sessions is very high. Your first session in the connection pool could serve about 80% of the requests (a typical real number for a 24/7 system with 500 users). If there is the slightest problem either with Oracle (memory leaks still happen especially in more OO-oriented modules, like XML) or your code, that session could consume a huge amount of resources.
   - Randomizing provides some protection from having a single very resource-intensive session, but it will make managing the total size of connection pool much more difficult.
3. Expiration mechanism (only for non-randomized connections) must be present, otherwise the size of connection pool will reach the high watermark and stay there. Even though it will save time for auto-extending, keeping sessions opened for unnecessarily long periods of time is very expensive, because this will lock a lot of database resources. PGA, UNDO, temporary segments, etc. are released only at the end of the session. This means that the faster you close sessions, the less resources are used at any one point in time

- A normal rule of thumb is 30-60 minutes of inactivity. Keep in mind that setting an expiration time to a couple of minutes may sound smart, but in the majority of cases, it is the wrong thing to do since you will encounter all of the problems of stateless implementation and, even worse, the connection pool will spend even more time opening/closing sessions.
- In addition to the expiration of inactive connections, there should be an expiration of heavy connections, such as when the size of the PGA allocated for the session exceeds some limit. This option is very nice for long-term projects, where you go through a number of different Oracle versions/patches/bug fixes because it is very difficult to discover a memory leak until the code is in production and there is no way out of it. The resource expiration mechanism gives you a nice back-door.

4. Full refresh: There should be a more civilized way of completely resetting all of the connections in the database other than bouncing the application server. It is recommended that you use a special type of request to the middle tier to stop it from serving an existing set of sessions (and eventually retire them) and get completely new ones.
    - The expiration mechanism is very handy if you need to modify some PL/SQL in a production system. Stateless implementations make people less scared of encountering an "existing-state-of-packages" error, but connection pools reintroduce this issue in most real environments.

Training Issues

There is a common mistake made by many people just starting to work with connection pools. They hear that the middle tier will reuse sessions instead of open or close them and they use all of the old tips and tricks that they know for client-server solutions. The danger is that this approach works perfectly fine in the development mode but creates complete havoc when somebody tries to add a second user. The reason for the problem is very simple. If you have a connection pool with only one user in the system, your code will always use the first connection (unless the pool is randomized), which is the definition of a stateful approach. But when you add a second user, in the majority of cases, requests from both logical sessions will be served by the same physical one and the previously perfectly working "client-server-ish" code will cause some very serious data cross-contamination.

One solution is not to tell developers about connection pools since they are an architectural way of solving resource workload problems on the system and normally should not have anything to do with development solutions. But as usual, there are exceptions. Sometimes, developers should know about alternative options for handling sessions. To illustrate, the following is a story from the author's experience.

The development environment was described as follows:

- PL/SQL wrappers on very complex Java classes, loaded into an Oracle database
- The Java code establishes a connection with the external geocoding server, passes data, and returns results.
- These requests are one of the most critical parts of the system and executed regularly by all users.
- The cost of the initial request is very high (up to 10 seconds) because of the whole initialization process (both Java and geocoding APIs), but additional requests in the same Oracle session take less than 0.3 seconds.

The Solution was to switch the whole application from a pure stateless solution to one that used non-randomized connection pools This made sense since the most costly request is the first request per session so the goal was just to keep the smallest number of sessions!

Session Resource Management

Another situation when developers must know about connection pools (and all code should be checked for compatibility) is in a stateless implementation when using session-level tricks inside of modules that would serve a single request. Normally, it is very convenient to use temporary tables of package variables as buffers while processing. since it is a built-in feature (because the middle tier would immediately release all of that when the session is closed). But when you start using connection pools, this is the number one cause of problems (even legal ones)! Since sessions are not closed anymore unless you do something about them, there is a high probability that one request could get data from the other one, causing yet another case of data cross-contamination.

This adds new meaning to the term "dirty data" since you cannot trust ANYTHING defined at the session level. Everything should be handled manually so that the built-in in the connection pool mechanism executes a special cleanup module (covers package variables, temp tables, both) before serving any request in the session, whether or not the request is intended to be on the server.

Packaged variables can be handled with a few lines of code as shown here. 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 to clean that data. The following routine could be somewhat useful (even if it is 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.

Another remote possibility is a join between V$SESSION and V$TEMPSEG_USAGE. This is known to cause very strange errors in some cases (including even ORA-600). However, the solution is simple. Just split the query in two as shown here:

```
procedure p_truncate
is
    v_exist_yn varchar2(1);
    v_saddr    v$session.saddr%type;
begin
    select saddr
    into v_saddr
    from v$session s
    where s.audsid = SYS_CONTEXT('USERENV','SESSIONID');

    select 'Y'
    into v_exist_yn
    from v$tempseg_usage u
    where u.session_addr = v_saddr
    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;
```

Database Resource Utilization

There is one core assumption underlying any implementation of connection pools: a single request to the database takes a very small amount of time. As a result, the total number of active requests at any point in time is small compared to the total number of logical users in the system. This means that the slightest slow-down in the processing of requests could very quickly kill the whole system. The worst part is that the system could work fine 99% of the time, but once some kind of a threshold is reached, the degradation spiral starts to unwind.

The more time needed to process an individual request, the more often it is necessary to add a new session to the pool. Because there are no free sessions, more simultaneous sessions cause more resources to be used making less resources available per session. This makes each individual request slower, and so forth. After a few cycles, the system has no resources left at all and collapses

These types of issues are very difficult to resolve in a production environment and should therefore be prevented using the following strategies:

1. The most often executed requests should be very carefully tuned because these requests define the average workload.
2. The most expensive requests should not enter the system via a connection pool at all. It is recommended that you completely avoid pooled sessions for any special kinds of requests.
3. The connection pool should notify administrators about reaching a defined workload level (e.g. allocated PGA per session or total allocated PGA) or number of sessions in the pool.

Conclusions

There is no way to build any reasonable web-based solution without going stateless, but there are different ways of doing that. Using or not using connection pools is not a matter of preference, but a matter of understanding exactly what are you trying to build. Every feature solves some problems and introduces other ones. It is your responsibility to balance the pros and cons of using connection pools. It is impossible to cover all of these in a single paper, but the main purpose here is 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.