ORACLE

Database Pool Sizing

November, 2026, Version 1.0

Copyright © 2026, Oracle and/or its affiliates
Public

Purpose statement

This paper is a continuation of the paper called “Application Programming Using Pooling” [1], and focuses on sizing the various pools that are used to control database resources. The primary focus is on Online Transaction Processing (OLTP) applications, although some aspects of batch processing are also covered.

While there is no single correct answer to the question “how should the pool(s) be sized?”, this paper discusses how typical configurations and application models impact both the *type* and *size* of pools.

A good understanding of the topics discussed in “Application Programming Using Pooling” is highly recommended.

Disclaimer

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle software license and service agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This document is for informational purposes only and is intended solely to assist you in planning for the implementation and upgrade of the product features described. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described in this document remains at the sole discretion of Oracle. Due to the nature of the product architecture, it may not be possible to safely include all features described in this document without risking significant destabilization of the code.

Copyright © 2026, Oracle and/or its affiliates / Public

ORACLE

Table of contents

Introduction **5**

Part One – Overview **6**

A taxi queue example 6

From taxis to databases 7

Pool sizing 7

Part Two – Pool Sizing **8**

Pool size parameters and statistics 9

A theoretical approach to calculating the pool size 10

Complicating factors 11

Application complexity 11

Application server counts 11

Variation over time 12

Networking differences 12

Service level agreements 12

Application statistics 13

Pool size limits 13

Handling overload 14

Dealing with complexity 14

Practical rules for pool size 14

Part Three – Actual Examples **16**

Brief example description 16

Injecting “faults” 19

Behavior with larger session pool 21

Wait for session instead of time out 23

Conclusions 26

Part Four – Statement Cache Sizing **27**

Part Five – Batch Processing **30**

Part Six – Summary **32**

Appendix A – References **33**

Copyright © 2026, Oracle and/or its affiliates / Public

List of figures and tables

Sample Code 1 8

Table 1. Typical pooling interface parameters. 9

Table 2. Typical pooling interface statistics. 9

Figure 1. Factor applied to core count vs FG idle percentage. 11

Table 3. Helpful application statistics. 13

Figure 2. 24h simulation; baseline. 17

Figure 3. Sessions in 24h simulation; baseline. 18

Figure 4. 24h simulation; timeline with injected faults. 19

Figure 5. Sessions in 24h simulation; timeout with injected faults. 20

Figure 6. Wait event graphs from Active Session History. 21

Figure 7. 24h simulation; timeout with increased session pools. 22

Figure 8. Sessions in 24h simulation; timeout with increased session pools. 22

Figure 9. ASH wait event graphs; timeout with increased session pools. 23

Figure 10. 24h simulation; wait with modest session pool. 24

Figure 11. Sessions in 24h simulation; wait with modest session pool. 24

Figure 12. 24h simulation; wait with large session pool. 25

Figure 13. ASH wait event graphs; wait with large session pool. 25

Table 4. AWR report statistics. 28

Table 5. AWR report showing parse counts per SQL. 28

Table 6. AWR report statistics for small statement cache size. 29

Table 7. AWR report showing parse counts per SQL for small statement cache size. 29

Sample Code 2 30

Table 8. References. 33

Copyright © 2026, Oracle and/or its affiliates / Public

Introduction

Oracle databases are the core of many applications, serving any type of industry, and covering a vast range of complexities and deployment sizes. The databases used can be anything from small databases to massive configurations with multiple Exadata servers, deployed on a vast size range of cloud or on-premises hardware. Such applications can support anywhere from a few concurrent users to millions of concurrent users. Each of these users can execute a large variety of transactions, including making telephone calls, doing online shopping, boarding airplanes, or paying for their goods in supermarkets. These systems are commonly known as Online Transaction Processing (OLTP) systems, although they often include batch-oriented processing as well.

The database workload varies in several ways: the number of users may change, the types and mix of activities may change, and the load from each user may change. In modern deployments, there are one or often more application servers that have pools controlling database or other resources, and proper sizing of these pools is an important aspect of system configuration.

“Application Programming Using Pooling” [1] describes the many different possibilities for pool configuration. This paper continues those discussions in six major parts:

1. Overview
2. Pool sizing
3. Actual examples
4. Statement cache sizing
5. Batch processing
6. Summary

Copyright © 2026, Oracle and/or its affiliates / Public

Part One – Overview

"Application Programming Using Pooling" [1] discusses the different concepts that are relevant for pooling. One of the most important concepts is that of an *authenticated database session*. In order to perform any work on Oracle Database, you need to be authenticated, which traditionally means a database username and password have been provided[1]. This both verifies that the user has access to the database and identifies a database schema with objects, such as tables. Additionally, when database transactions occur, e.g. using insert, update or delete to manipulate data, these transactions are done as part of a database session.

When some part of the application code is ready to perform database work, it initially requests an authenticated database session—often called "borrowing" a database session. It then performs the actual database work by executing some SQL statements and possibly a commit, and it finally finishes by returning the borrowed database session. An application written in Java is a very common example. Each Java process has multiple threads running a few different categories of code: worker threads that do what is described, threads that handle the user interface, and housekeeping threads controlling the pool of authenticated database sessions. An example of the latter is Oracle's Universal Connection Pool (UCP) [7].

This approach, where an application explicitly borrows a database session while it does database activity and returns it immediately after, has the main benefit of allowing a larger number of worker threads to share a smaller number of pooled database sessions. This implies that the sessions in the pool can be shared and effectively utilized, and that there is a mechanism to control the number of sessions in the database. Over time, these sessions are either idle or borrowed by some worker thread.

Note that this example uses the term "worker threads" as that is the actual implementation using Java and UCP. There are other models, but the distinction between them is only important for the discussion in "Application Programming Using Pooling" [1]. In the rest of this paper, the simple term "worker" is used when the implementation details are not relevant.

A taxi queue example

In order to understand some of the topics around pool sizing, let us initially use a thought experiment: You fly into an airport, and need a taxi to bring you to your final destination. After leaving the arrivals hall at the airport, you follow the signs to the taxi rank and realize there is a long taxi queue. Is that a nice situation?

The answer depends on what is meant by "taxi queue". If it is a long queue of taxis waiting to serve customers, you will likely find the situation nice, because you can quickly get a taxi. If, on the other hand, a taxi queue is a long queue of customers waiting for taxis to arrive, you are also likely to wait a long time before you can get a taxi. There is also the opposite aspect: for a taxi driver arriving at the airport to pick up new customers, an already long queue of taxis means the driver will be waiting a long time for a customer. Unless the taxis arrive at the airport at exactly the same rate at which passengers arrive, there will always be a "taxi queue", which is either a queue of taxis waiting to get customers or a queue of passengers waiting to get a taxi.

We can also take this example further: it is likely that the roads leading away from the airport have limited "throughput" —a limitation to the number of vehicles per time unit. In this case, adding more taxi lanes or adding more taxis doesn't improve the overall throughput. A delay getting to your final destination may be caused by too few taxis, so you need to wait to get one, or by too many taxis, so your taxi is delayed due to congesting traffic.

A third aspect of this example is how the customer or the taxi driver reacts if the queue on their respective side is too long. Upon seeing the long queue, either can decide to do something else: the taxi driver may simply drive somewhere else in the hope of getting a customer, and the arriving passenger may decide to use a different means of transportation or have a meal at the airport in the hope the queue will be shorter later on.

[1] More secure authentication methods like OAUTH or IAM tokens are available; this distinction is not relevant for the discussions here.

Copyright © 2026, Oracle and/or its affiliates / Public

From taxis to databases

In many aspects, database servers or, more precisely, session pools work the same way: they can be almost always idle if there are no arriving requests to process, or they can be constantly busy with growing queues if there are more arriving requests than there are pooled sessions to handle them. Similarly, the session pool can be so large that congestion in the database is the factor limiting throughput.

Pool sizing is all about finding the right balance between requests arriving, requests being processed, and preventing congestion in the database.

There are also analogies to the customer's or taxi driver's decision to do something else. The database analogy to the long queue of taxis is far too many resources, which could be adjusted downwards to better match the requirements. The application analogy to the long queue of passengers is to display a message on a website saying "Our system is too busy; please try later.".

Pool sizing

Let us briefly introduce some of the considerations about sizing the pool or pools. The database ultimately needs to process all SQL[2] and all transactions. However, the database has a finite number of resources, and that limits the number of requests it can handle. The actual limit depends on many factors, particularly the amount of computing resources and the complexity of requests. Additionally, in OLTP systems, these limits are typically not hard limits and instead manifest gradually. If the database is brought gradually towards its operating limits, the first effect is that transactions gradually take longer, and if the workload is increased further, there is a risk that the database becomes unresponsive. This often causes massive failures or time-outs for application users.

In an over-simplified example, assume all requests are identical, assume they each take 75ms with a session borrowed from the pool[3], and assume the database can handle 160 requests every second. If we multiply these figures, we get 160/s * 0.075s = 12, so a session pool size of 12 is appropriate. If all entries in the pool are constantly busy, each will do 1/0.075s or slightly above 13 requests per second, so all 12 will do the 160 that the database can handle. If the pool size were larger than 12, there would be a risk of overloading the database beyond its capacity, and if the pool size were smaller than 12, there would be a risk of requests waiting in a queue even if the database could have handled more.

It is important to point out that in the context of this paper, the discussion is focused on OLTP processing where the arrival of requests is *random*, and each request arrival is *independent* from the arrival of the previous or next request. Batch processing, and the vast differences from OLTP, is discussed in Part Five of this paper.

[2] We use the term SQL throughout this paper to refer to work done by the database. Generally, the database also executes other elements such as PL/SQL, stored procedures, and database commit or rollback.
[3] The 75ms is the time spent while the worker has borrowed a session from the pool. During this time, there will be time spent in the database, in the network, and in application code. In Part Two, we will discuss this aspect in detail.

Copyright © 2026, Oracle and/or its affiliates / Public

Part Two – Pool Sizing

Applications are almost always a combination of many different database requests. Some may be as simple as just executing one or two simple SQL statements, others may execute many different SQL statements. Some SQL may, in itself, be complex and other requests may include DML and transaction processing.

The typical pseudo code for an OLTP application might be:

Sample Code 1

```
handle=identify();
loop ...
  user_interface();
  application_logic0;
  get_session(handle);
    sql1;
    application_logic1;
    sql2;
    application_logic2;
    sql3;
    application_logic3;
    commit;
 release_session(handle);
end loop;
```

Initially, the application authenticates itself with proper database credentials, and it subsequently goes into a loop that has these major parts:

1. Waiting for an actual request from an application user and performing some application logic without access to the database. The time spent waiting cannot be predicted, but you can typically predict the arrival rate, which is the product of the total number of workers executing this code times the reciprocal of the average time between each request. In other words, *arrival_rate = worker_count / average_time_between_requests*.
2. When ready to perform database work, it will get (borrow) a session from the session pool.
3. The necessary SQL and possibly other database calls, such as `commit`, are performed *without any waits* for application users.
4. There will sometimes be application logic executed while the session is held.
5. The borrowed session is released back to the pool. After this step, the application no longer has any database state available.

If we compare to the simplified case in Part One, where it was assumed each request takes 75ms, we need to be more thorough in how and where time is measured. In Sample Code 1, the 75ms could be roughly the time spent from when we start borrowing the session until completion or releasing the session. If there is an available session in the pool, the call to get the session will be fast, so most of the 75ms is spent while having the database session. This time is the combined time of the following:

1. Application time to do various preparations.
2. Time spent on the network between the application and the database.
3. Time spent in the database.
4. Application time spent processing any data received.

We need to distinguish between time spent in the database to time spent outside the database, and none of 1, 2, or 4 are “free”. These all take time, and therefore, the 75ms we previously assumed per request is necessarily less than 75ms of database time. If the database transaction is just a single, complex SQL statement (or PL/SQL block) with little data being sent between the application and the database, most of the time may be database time.

Copyright © 2026, Oracle and/or its affiliates / Public

Conversely, if the request executes a large amount of very fast SQL, most of the time may be spent on the network or in processing in the application with only a little time spent in the database.

All these factors impact pool sizing.

Pool size parameters and statistics

Different pooling technologies have different parameters. The following table lists the typical parameters that exist in most pooling interfaces.

Table 1. Typical pooling interface parameters.

Parameter	Description
Minimum pool size	Sets the minimum allowed number of sessions in the pool and therefore the minimum number of connections and sessions in the database[4].
Maximum pool size	Similarly, sets the maximum allowed number of sessions in the pool.
Inactivity timeout	When the current size of the pool is larger than the minimum, and a session in the pool has been inactive for some period (i.e. that session has not been borrowed by any worker), it will be disconnected from the database.
Maximum lifetime, maximum reuse count	Each session in the pool may be forced to disconnect after having been connected to the database for some time or having been borrowed a number of times. Typically, this disconnect implies a new database connection will be established.
Inactive use timeout	If a session is borrowed from the pool but the worker does not actively use it for some time, the worker receives an error.
Borrow settings	When a worker executes the request to get a session, various parameters control how that request is handled in case there are no free sessions available. The worker can simply wait until a session is available or created in case the pool has not yet reached its maximum. The request may also time out, either because no entry becomes available or because establishing a new database connection takes a long time. The precise details of this are often specific to each pool technology.

In addition to the configuration parameters for the pool, there are often statistics values available in the pooling interfaces.

Table 2. Typical pooling interface statistics.

Statistics	Description
Open session count	The number of sessions in the pool that have an established connection and session in the database. The value is between the minimum and maximum pool size.
Busy session count	The number of sessions that are currently borrowed by some workers. This number is, at most, the value of open session count.[5]
Other statistics	There may be further statistics about number of failed attempts at getting a session, time spent waiting, overall total of database sessions established and more.

[4] This is assuming dedicated connections in the database. DRCP or Shared Servers are more complex in this respect.
[5] Note that the term "Busy session count" that is used by Oracle Call Interface and other APIs refer to the fact that the session is busy or active as seen from the pool perspective. This is what we refer to as "borrowed" in this paper. The session may not actually be active from the database perspective.

Copyright © 2026, Oracle and/or its affiliates / Public

There is variation in the actual implementation in the various interfaces such as Oracle Call Interface [5] or UCP [7] and you need to consult the documentation for your actual pooling interface.

It is crucial for system administrators to have access to a real time monitor of both the open and the borrowed session counts, and to have the ability to (easily) change the actual values for the minimum and maximum pool sizes.

A theoretical approach to calculating the pool size

In [2], Toon Koppelaars discusses what impacts the needed pool size and presents a formula that can be used to find the proper pool size. While the formula is theoretical and has some conditions that, for all practical purposes, cannot be met, it still is a very good introduction to the concept.

A brief summary of the presentation follows:

We assume all processing on the database side is CPU and we assume the hardware is fully scalable up to the actual number of cores[6]. In different terms, we assume the database can provide a certain number of database CPU seconds every clock time second. This is called the core count or #cores in [1].

We assume a fixed application that strictly follows sample code 1 above, including running some application logic between the different database calls while the session is borrowed.

We then *measure* the following figures:

- The total time spent on the application side from completing the `get_session()` call until the `release_session()` call. This is the time during which the session is borrowed from the pool—the *reservation time* in [2].
- The time spent in the database doing the SQL (and other operations such as `commit`) on behalf of an application request, or, practically, the database CPU spent doing SQL (and other work) executed between `get_session()` and `release_session()`.
- The difference between the reservation time and this database time is the foreground idle time or *FG idle time during reservation*. It is called *idle time* as the database is idle while the session is still borrowed by some worker.

The necessary pool size is then given by:

$$\frac{reservation\ time}{reservation\ time - FG\ idle\ time} \times \#cores$$

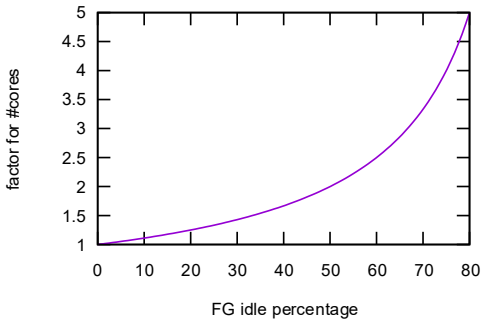
The FG idle time will always be less than the reservation time. Ideally, the difference between the two should be as close to 0% as possible, meaning that the database is continuously active in the period between `get_session()` and `release_session()`.

[6] The number of cores, or the often-used number of CPUs, can typically not be easily related to the physical core count of the database hardware. At the hardware level, most computers use multiple threads. These threads are then mapped to cores or CPUs for the Virtual Machines that run the actual database code. Further, different models for providing and managing compute resources exist such as Elastic CPU and OCPU and resource management. For the discussions in this paper, cores therefore are a loose representation of the CPU resources available to the database.

Copyright © 2026, Oracle and/or its affiliates / Public

If we plot the factor applied to the core count vs the FG idle percentage, we get:

Figure 1. Factor applied to core count vs FG idle percentage.



The foreground idle percentage depends primarily on these two factors:

- Time spent on the network, which again depends on the number of SQL statements and the latency between the application server and the database server.
- Time spent in the application code processing data.

Complicating factors

The formula in the previous section has several conditions that are unlikely to be met in real production cases. Some of the important factors impacting actual session pool sizing are application complexity, application server counts, variation over time, and networking differences.

Application complexity

Database applications do not consist of one single well-defined business transaction. Rather, there are many different types of transactions that execute different SQL and require different application logic. As a result, both the reservation time and the FG idle percentage can have large variations. The arrival rate of the different business transactions will also normally be very different.

As an example, an application serving a web shop may frequently execute product searches, less frequently execute an “Add to cart” transaction, and even less frequently execute a transaction that charges a credit card or prepares an order for shipping.

Application server counts

In the discussion so far, exemplified by sample code 1, we are effectively discussing a single Operating System process that runs many worker threads executing business logic that borrow database sessions as required. In actual deployments, it is normal to have multiple application servers, each running multiple processes, each with their own pool. As a result, the total number of sessions in the database may potentially be as high as the product of the count of application servers, application server processes per server, and maximum pool size per process. This product can easily be so high that there is a potential risk of overloading the database.

If the application uses a microservices model, this product may be even higher as each microservice needs to have its own pool with a minimum and maximum pool size.

Copyright © 2026, Oracle and/or its affiliates / Public

Variation over time

Many applications have variations over time that can be of different nature:

- Activity may be more frequent during business hours with much less activity outside those hours.
- Activity scheduled to run outside normal business hours may be of a very different nature and it may run nightly, weekly, monthly, or annually. See Part Five – Batch Processing of this paper for a discussion about batch processing.
- Systems supporting a 24*7 environment may have differences due to time zone variations.
- Unforeseen events, both coming from the outside and from various places inside the system, may happen.

If time variations are combined with application process or server variation, you may very well have a system where the load on the database changes over time from one set of servers and processes to another set of servers and processes and/or between different types of services or servers. These variables frequently impact session pool sizing and often make session pool sizing complex and variable over time.

Networking differences

One of the factors that impacts session pool sizing is the network latency because higher network latency means increased FG idle time. Although application servers ideally should be as close as possible to the database servers to reduce the latency, this is, for many different reasons, not always possible. Additionally, different application servers may have different latency to the database server.

If the complete system includes high availability with different geographical locations, failure in one component with a subsequent move from a primary to a secondary site is likely to also imply a change in network topology and, therefore, latency.

Service level agreements

There are often two types of requirements for OLTP type applications: overall throughput measured in transactions per time period, and service level agreements stating the highest allowed response time for a certain percentage of transactions. The following is just one example of these requirements:

- The system must support a rate of 15,000 transactions per second
- 95% of the transactions must be 20ms or faster
- 98% of the transactions must be 30ms or faster

As we shall see later, the two types of goals are contradictory: you can maximize the throughput if there are no requirements for response time, or you can get the best possible response time by accepting a lower throughput. However, you *cannot* have a response time goal for 100% of the transactions; there is always a risk that some (albeit few) transactions will have a very high response time.

Copyright © 2026, Oracle and/or its affiliates / Public

Application statistics

When applications are executing code exemplified by sample code 1 above, there are several statistics that are important or useful to have available:

Table 3. Helpful application statistics.

Statistics	Description
Wait time to borrow a session	The first part of the loop is the `get_session()` call, which in many cases executes quickly as a session is typically available in the pool. However, when a session is not available, the call takes time to either wait for some other worker to return its borrowed session to the pool or for the pool itself to increase by establishing another logon to the database. The wait time is an important statistic.
Total processing time	After a session has been borrowed from the pool, the total time of all application logic, network time, and all database activity (primarily SQL and commit) appears as the processing time[7]. The sum of this time and the time taken to borrow a session is relevant for the service level agreement. The total processing time is effectively the same as the reservation time used for the theoretical session pool sizing discussed above. Processing time is also an important statistic.
Database time	As explained in [2], processing time is really the sum of three times: application time, network time, and database time. To fully perform end-to-end performance analysis, the time taken by all three components should be available.
Network time	The usefulness of network time is often underestimated, but when available, end-to-end performance analysis is better.

In general, the more statistics that are available, the more precise performance investigations can be made with subsequently better configuration settings.

Pool size limits

During startup of an application server (process), the application server also creates sessions in the database according to the minimum pool size. During execution, the server may create new sessions or drop existing ones as the pool size can vary between the minimum and maximum. Establishing a database session—that is, to log on to the database—is several orders of magnitude more resource-consuming than executing the typical SQL of an OLTP application, and the rate of database logons should therefore be limited.

A common debate is whether session pool sizes should be variable or fixed. Except in very specialized applications, it is not feasible to have fixed pool sizes, and therefore not feasible to completely prevent database logons after those done initially. The primary reason for this is the various types of complexity discussed earlier. A goal of configuring the pool is to find a balance between all the following:

- The maximum size of all pools can be directly mapped to the maximum number of database sessions, which must be limited to make sure the database is always running within its safe operating limits.
- The minimum size of the pools determines the number of database sessions that start when each application server (process) is started.
- The inactivity timeout impacts how quickly unused sessions are disconnected from the database and therefore also implicitly how many and how quickly sessions are created in the database when the overall load increases.

[7] Note that this paper *only* discusses the application server and the database server. The whole stack naturally also includes front-end servers handling the user interface. As a consequence, the real total processing time also includes network between application server and front end, front-end processing, and finally network between the front-end server and the application user's device.

Copyright © 2026, Oracle and/or its affiliates / Public

- Over time (e.g. day/night or weekday/weekend), the actual activity may move between application servers or processes.

Handling overload

If a system gets overloaded, the primary concern is to protect the database, which means the system must be able to deal with the overload *before* it reaches the database. This implies some users are impacted negatively. The alternative is that the database itself gets overloaded, which implies a high risk of making the whole system unresponsive. There are at least two ways to react to an overload situation:

1. Requests are simply put in a queue and handled whenever the system is capable of doing so.
2. Requests are cancelled or timed out and the user is informed about the situation.

Both methods are found in actual systems, often in a combination.

Dealing with complexity

There is no single correct answer as to how to manage all these complex cases. Eventually, the actual deployment and configuration of the many components is a trade-off between risks of outages, system responsiveness, and database utilization. At one extreme, you can set the size of all session pools so small that the database is guaranteed to always run within its safe limits, even when all pools are utilized to their maximum. However, this implies the application responsiveness is very low since users must frequently wait for a session even if the database has ample resources. The other extreme is that each pool is allowed to be so large that a single application server or application server process by itself can bring the database to its safe operating limit. In that case, there is a very high risk of outages due to database overloading.

Practical rules for pool size

The two primary configuration parameters for session pools are the minimum size of the pool and maximum size of the pool, and the two primary statistics are the actual number of open sessions and borrowed sessions. The number of open sessions increases up to the maximum size as demand increases and decreases down towards the minimum size as demand decreases as controlled by the inactivity timeout parameter.

Ideally, the sum of the maximum size of all pools should be limited by the capacity of the database so that the database does not get overloaded if all pools are utilized to their maximum. Reaching this is, in most cases, not possible simply due to the typically high number of application server processes, each with their own pool, in combination with the many types of variation mentioned earlier.

For the individual pool, however, we can find a reasonable maximum size:

The maximum size of the pool should be between the typical counts of borrowed and open sessions in the pool during ordinary high activity.

This is because even with steady high activity, the number of borrowed sessions is clearly smaller than the number of open sessions. The random nature of arriving requests causes workers to be in different states: in the process of getting a session, executing business logic, in some network, or active in the database. If the maximum size of the pool is only slightly larger than the typical borrowed session count, it is more likely that workers must wait while getting a session. As the maximum size increases, the likelihood that this wait takes place and, subsequently, the actual time spent waiting decreases, and if the maximum size reaches the typical open session count, there is practically no wait. If the maximum size is increased further, the waiting time does not decrease but the risk of overloading the database does increase. Therefore, the maximum size should never be larger than the steady state open session count.

In the low range, we can also state requirements for the minimum pool size:

The minimum size of the pool should be lower than or just at the typical borrowed session count.

Copyright © 2026, Oracle and/or its affiliates / Public

In many cases, keeping the minimum at zero is reasonable, particularly if workload variations up or down are rare. Due to the nature of the pool, where an actual logoff only takes place after the idle timeout expires, the pool tends to adjust itself to a reasonable size. If logoff and, therefore, logon appears to be too frequent, you can increase the minimum size to be close to the typical borrow count. Setting it higher than this value may make the pool adapt more quickly to sudden increases, but it also implies high logon rate when the application server (process) is started. Additionally, always having (many) open sessions that are not borrowed increases the risk of a sudden increase in activity, also known as an *activity storm*. These can be as troublesome as connection storms.

Note that an implication of the two rules is that you should never set the minimum and maximum pool size to the same value. The values should always be balanced between the typical borrowed and typical open session counts. Another implication is that you cannot find reasonable values for the minimum and maximum pool size unless you have instrumentation showing the typical borrowed and open counts. Such instrumentation *must* be part of your application design!

In Part Three, we will use actual examples to show changes in behavior with different pool sizes and different uses of pools.

Copyright © 2026, Oracle and/or its affiliates / Public

Part Three – Actual Examples

In this part, we will present the details of an actual workload example used to show how the many variables have an impact on the behavior of both the database, the application, and ultimately the application users. The example is exactly that: an example. The results provided are, therefore, specific to that example. The actual figures cannot be extrapolated or used with any real-world application, but they can be used to understand the complexity of real-world applications, and to understand how different configurations impact the overall system behavior.

Brief example description

The example that will be used throughout consists of these major hardware and software parts:

- An Autonomous Transaction Processing database configured to allow a maximum of 32 CPUs/s.
- A virtual machine acting as the application server with sufficient resources to ensure that the machine is never the bottleneck.
- An emulated OLTP workload based on the RWP*Load Simulator [3].

The workload is emulating a mix of some ten different “business transactions”, some of which are similar to real transactions and others that are completely artificial. To get results for this paper, the workload is configured to simulate a full 24h processing with day/night changes, combined with shorter periods of higher throughput and injection of two types of “faults”. The purpose is to have a workload that is somewhat realistic: the short periods with higher throughput emulate sudden *external changes*, such as higher activity at a certain time of the day, and the “faults” emulate sudden *system changes*, such as periods with higher I/O times or higher database activity not caused by external changes. Both of these are explained in detail below. During the test, each 24h simulation is a run that lasts 10 minutes. During those 10 minutes, the workload is initially low, emulating night time, then increases to a maximum, emulating day time. The workload stays at the maximum amount for some time, and toward the end of the simulation, decreases again.

When running, several processes start, and each of these processes run several worker threads that execute the emulated business transactions randomly, arriving at a specified rate. During execution of an emulated business transaction, the process borrows a session from a session pool.

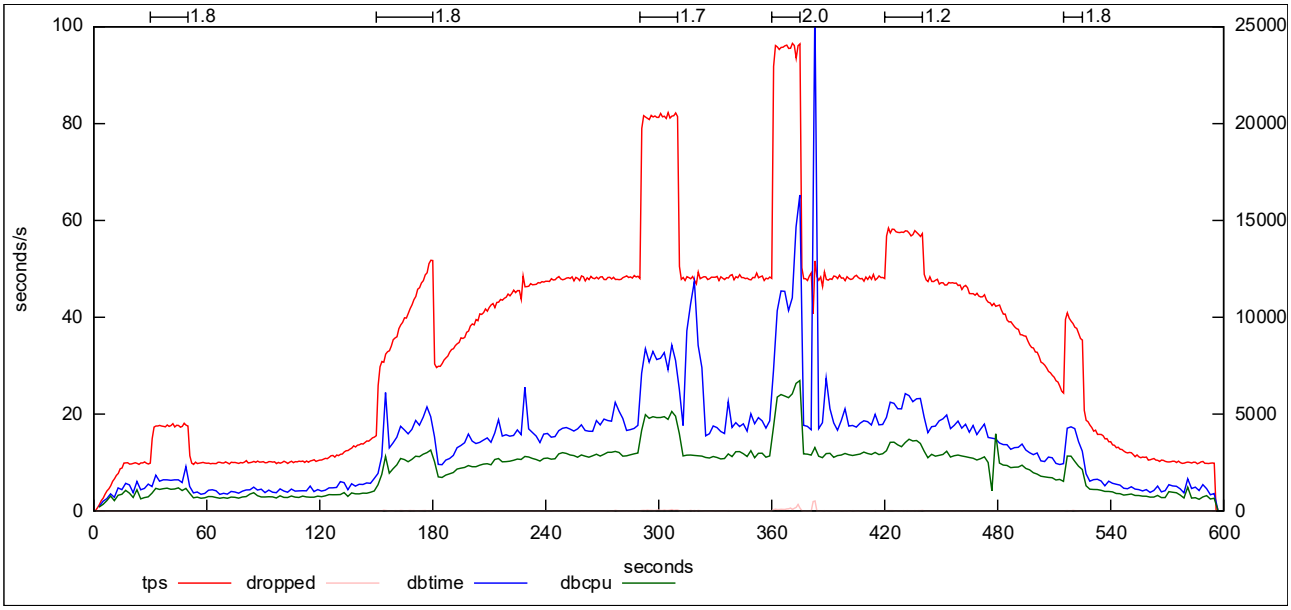
The default standard case uses the following values as a baseline:

- The number of processes is 25.
- The session pool in each process is variable, between 3 and 10 entries, with the inactivity release time set to 10s.
- If a request to borrow a session cannot be fulfilled within 100ms, the request is cancelled.
- The number of worker threads is 50 per process.
- In the normal period, when daytime is simulated and outside the shorts period with sudden external changes, each worker thread simulates an arrival rate of 9.6 transactions per second (tps). With 50 worker threads, this is 50*9.6 tps = 480 tps per process, and with 25 processes, this is 25*480 tps = 12.000 tps total.

Copyright © 2026, Oracle and/or its affiliates / Public

The following graph explains the details of a run that includes the emulated night/day intervals and the periods with higher workload due to external changes:

Figure 2. 24h simulation; baseline.

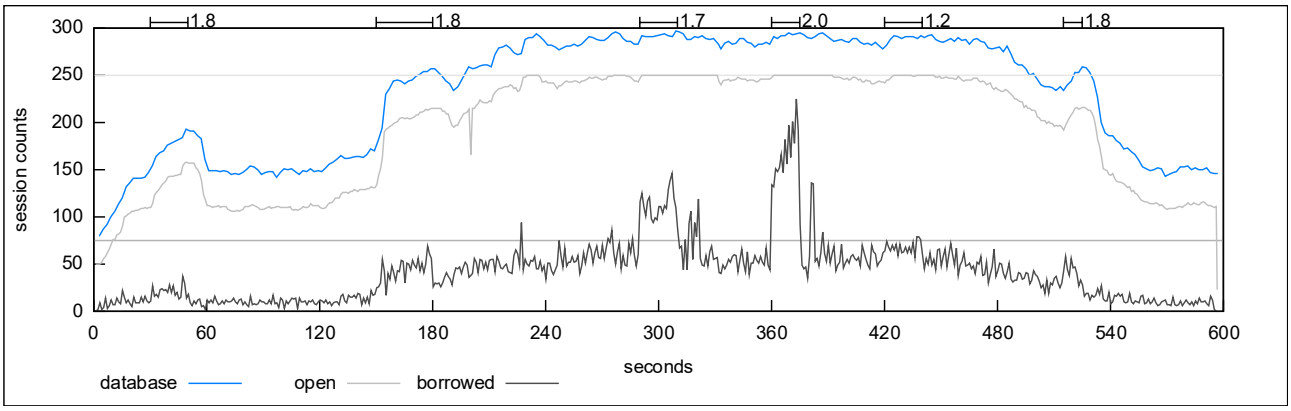


- The x-axis going from 0 to 600 shows the runtime in seconds, which is used to simulate a full 24h period.
- The red curve with values on the right y-axis shows the overall throughput measure. The period until 120s and again at the end of the run is the emulated night, the period from 240s until 430s is the emulated day (with the normal throughput of 12.000 tps mentioned above), and the periods between respectively ramp up and ramp down.
- In the six periods marked with black above the x-axis, the workload is increased by a factor shown above the graphs. As an example, from 290s to 310s in the run, the factor 1.7 is applied, and the throughput reaches 22.100 tps.
- In cases discussed later, there are similar blue and red markers under the x-axis showing periods when the workload is injected with “faults” that emulate *system changes*.
- The green and the blue curves, read against the left y-axis, show, respectively, database cpu and database time.
- The pale red curve, read against the right y-axis, shows the number of requests per second that are cancelled due to timeout while waiting for a session in the session pool. In the graph above, there are some barely visible at the largest peak, approximately 380s into the run.

Copyright © 2026, Oracle and/or its affiliates / Public

Part Two mentioned that session pools have two important statistic values: the number of open sessions and the number of borrowed sessions. These are shown in the following graph, which has an x-axis matching the previous graph.

Figure 3. Sessions in 24h simulation; baseline.



In this graph, the different session counts are:

- The horizontal pale grey lines at 75s and 250s show the range of the session pool, which matches the 25 processes' times respective minimum and maximum size of 3 and 10.
- The dark grey curve shows the number of borrowed sessions totalling 25 pools, as reported by the session pool.
- The light grey curve shows the number of sessions that are open in the database, also reported by the session pool.
- The blue curve at the top shows the number of established connections as reported by the database. This curve is somewhat higher than the number of open sessions reported by the session pool as there are some additional sessions outside the session pool. The details of these additional sessions are not relevant for the discussion in this paper.

We can make the following observations from these two graphs:

- In the first two and the last periods, where the workload is increased by a factor of 1.8, there are extra sessions created in the pools, and the database immediately reacts with an increased DB CPU and with the DB TIME, except for a brief peak, still being practically the same as DB CPU. This is the ideal situation, where the workload causes no extra waits in the database.
- Shortly after the workload returns to normal after the sudden workload increases, the size of the pools is reduced due to the inactivity time-out.
- The number of borrowed sessions in the pools are, during most of the run, much lower than the number of open sessions in the pool. This implies the system is running such that requests for a session from the pool are, in most cases, fulfilled right away and without waits.
- When the emulated day is reached, some 240s into the run, the pools are almost at their highest capacity as seen by the light grey curve, which almost reaches the pale grey line at 250 sessions. This means that the pools effectively prevent creating further database sessions.
- In the third and fourth periods, where the workload is increased by a factor of 1.7 and then 2.0, the number of borrowed sessions in use increase. DB CPU increases as expected, but there is clearly extra DB TIME, i.e., extra waits in the database. The nature of these waits, found using Active Session History, is the "log file sync" wait event. A detailed inspection of application-side timing also shows that in these periods, the time taken to wait for an available session increases from less than 0.1ms outside these intervals up to 4ms during the period with the doubled workload. Therefore, and as conjectured in Part Two, when the workload reaches the

Copyright © 2026, Oracle and/or its affiliates / Public

highest possible throughput, the response time increases. Conversely, the smallest possible response time is only possible outside the periods with high throughput.

If we compare these observations to rules for pool sizing (see Practical rules for pool size), we can make the following statement: when the number of *borrowed* sessions grows towards the maximum number of sessions, there *will be waits* to get sessions from the pool. This will be very visible if the number of the borrowed sessions is constantly at the maximum size, as the majority of requests then must wait. Several examples of this are shown later. Since this run is configured to use time-out in getting sessions, there are also requests cancelled due to this. Had this been a real application, the application user would have received an error saying something like “The system is too busy right now”.

It is often argued that, in such a case, the maximum size of the session pool should be increased, and that long waits for requests or “too busy” errors are unacceptable. However, that argument is incorrect: if we allowed a larger session pool, there would be a high risk of overloading the database, which would impact the entire system. With the modest maximum session pool and the potential for time-out while getting a session, the database stays at its safe operating limits, and only a minority of users are impacted.

Injecting “faults”

In order to show the effects on pools and on the database, we also inject two types of “faults”:

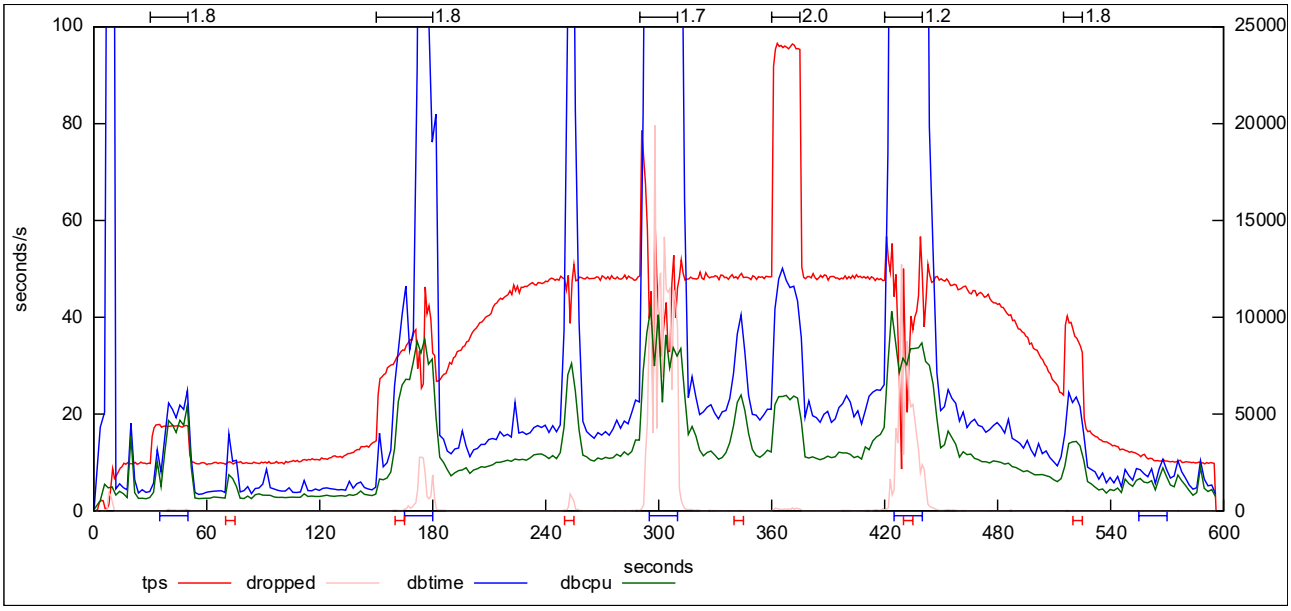
- Make some SQL statements use literals rather than bind variables (placeholders), causing hard parse to take place.
- Make some queries that normally use an index to lookup rows use a full table scan instead.

These two types of injections show how anomalies or database side effects can impact behavior. In actual cases, there is a much broader type of database side effects that can cause unexpected behavior, so these two are simply used as examples.

We will later show how typical configuration changes to application side pools have an impact on how the database ultimately deals not only with external changes, but also how it deals with unexpected internal (database-side) changes.

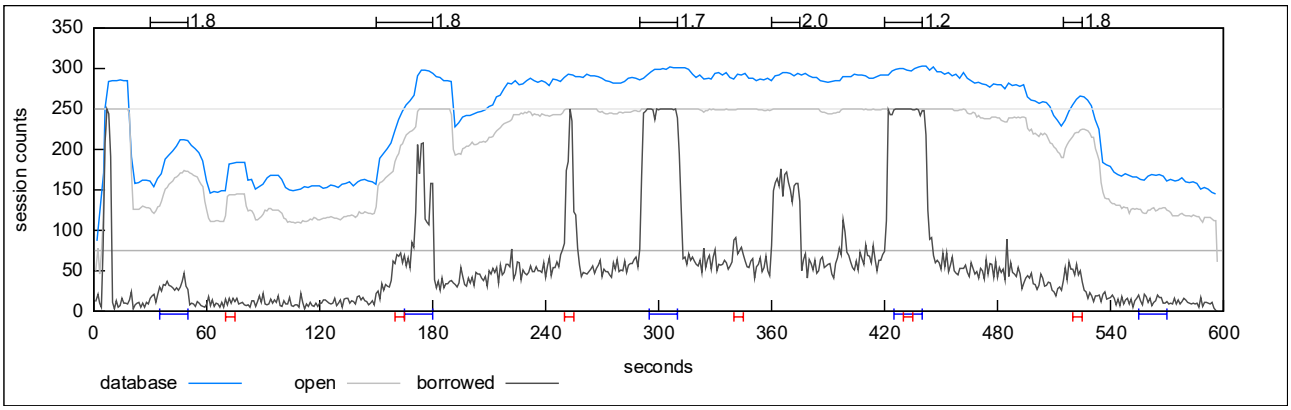
Including these two injected faults, a run now shows results like these graphs:

Figure 4. 24h simulation; timeline with injected faults.



Copyright © 2026, Oracle and/or its affiliates / Public

Figure 5. Sessions in 24h simulation; timeout with injected faults.



The red, pale red, green, and blue lines still represent throughput and cancelled requests against the right y-axis, DB CPU and DB TIME against the left y-axis, and the 600s runtime emulating 24h night/day variation against the x-axis. The intervals marked with black are, as previously, showing periods with higher workload. Additionally, the injected "faults" are:

- The blue markers just below the x-axis show periods when the workload is injected with full-table scans of large tables that are normally read using an index scan.
- The red markers just below the x-axis show periods when the workload is injected with SQL statements using literals, and therefore hard parse, rather than using bind variables.
- These injections may overlap with each other and with periods of higher throughput.

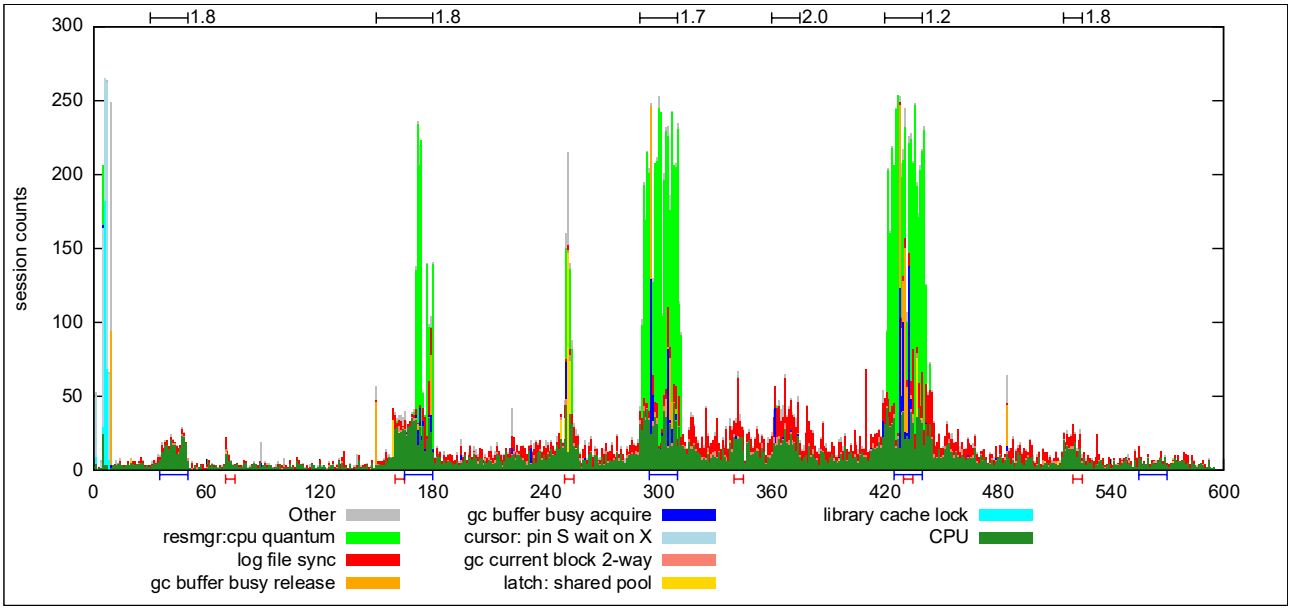
Since we now have both external and internal factors impacting the overall load, there are new things to observe:

- The injections do have an impact on the database, which shows much higher and more visible waits. This is demonstrated by the blue curve (DB TIME) in the first graph.
- Database CPU (green in the first graph) also increases, bringing the database close to its safe operating limits.
- Generally, the database does get through the periods with injected faults and gets back to a normal state as soon as the faults are over.
- On the application side, there are now visible requests that get cancelled, which means users get a notification like "system is too busy". However, as we see later, the alternative of always waiting until a session is available has negative side effects.
- Note that through the worst periods, 290s-310s and 420s-440s, the number of borrowed sessions in the pool reaches the maximum pool size. So while this implies longer overall execution time, it also helps keep the database safe.

Copyright © 2026, Oracle and/or its affiliates / Public

Let us also understand why the database—despite the injections and despite being at its safe operating limits—still has reasonable behavior. A look at the wait event graphs from Active Session History (ASH) explains this:

Figure 6. Wait event graphs from Active Session History.



In addition to CPU (dark green), there are two dominating wait events: log file sync (red) and resource manager (bright green). The log file sync event is normal and expected as there is a high rate of DML, and the resource manager waits are working as designed as they are setting an upper limit to the use of database CPU.

In summary:

- In normal circumstances, without injected faults and during sudden workload increases, the database is fully capable of keeping up with the workload.
- The number of borrowed sessions in the session pools is mostly less than ¼ of the number of open sessions, implying there are low waits to get a session.
- During sudden increases, the number of borrowed sessions grows toward the number of open sessions, which implies there *will be waits* for sessions. This is the *correct* design as it helps keep the database safe.
- When faults are injected, there are impacts on the application side, including long wait times to get a session, cancelled requests, and longer time spent in the database. However, the modestly sized session pools mitigate the overall impact, and when the fault is no longer present, the overall behavior quickly returns to being stable and normal.

Behavior with larger session pool

It is frequently argued that the upper size of the session pool should be set sufficiently high to guarantee a session is always available for request. However, keeping the session pool modest is almost always the correct thing to do. To show the consequences of using a larger pool, the previous run is repeated with only one change: the maximum size of session pools is increased from 10 to 30 per each of the 25 processes, implying up to 750 sessions in the database.

Copyright © 2026, Oracle and/or its affiliates / Public

The results are shown in these graphs:

Figure 7. 24h simulation; timeout with increased session pools.

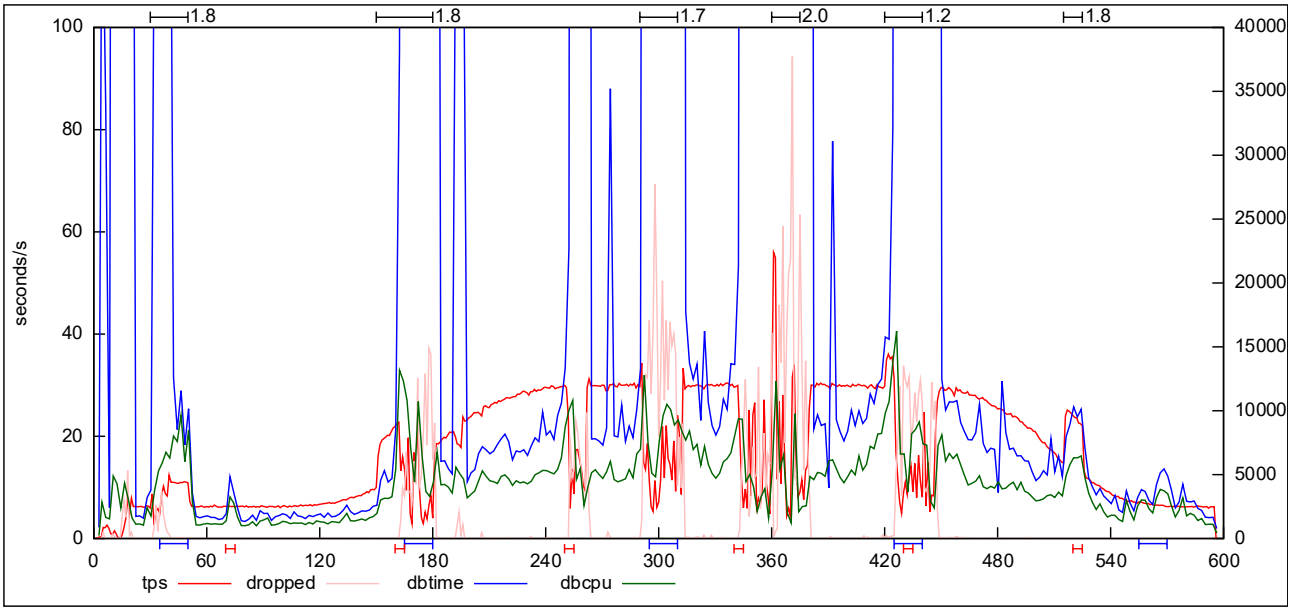
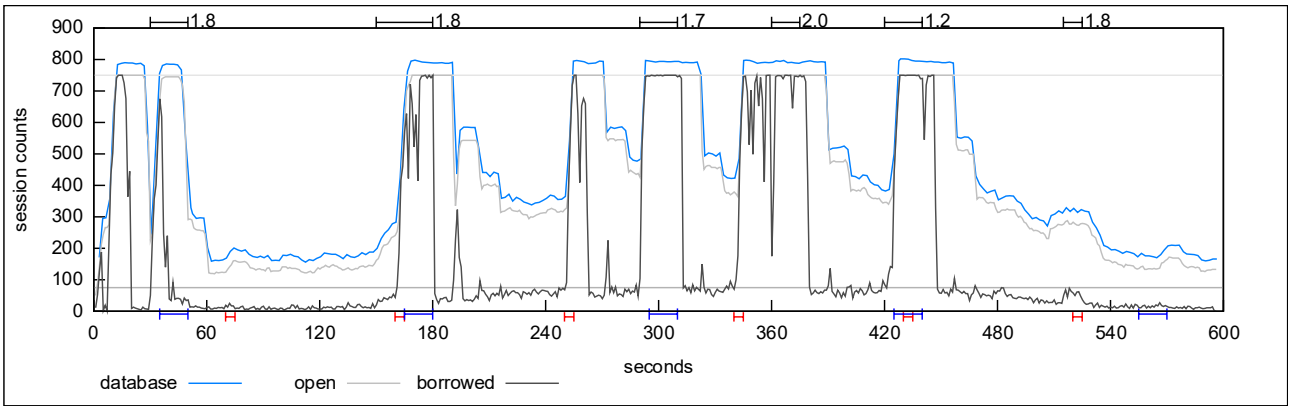


Figure 8. Sessions in 24h simulation; timeout with increased session pools.



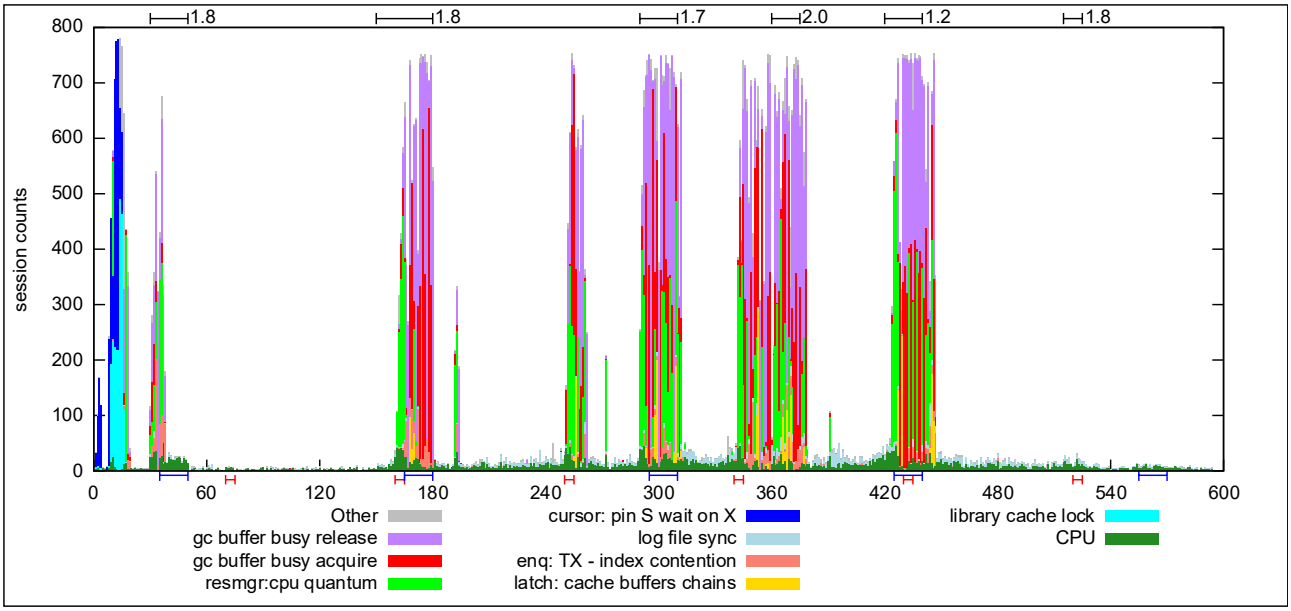
Some points worth noticing are:

- The number of requests that are cancelled *increases* during the periods with injected faults; this is exactly the opposite of the expected outcome.
- The period during which the database, and therefore the application, is impacted by the faults continues even after the fault is no longer there, and the faults have a far more visible impact. In particular, the red fault (full table scans) lasting from 340s-345s into the run has implications for almost 30s with very low throughput, a high count of cancelled requests, and generally very poor database behavior.

Copyright © 2026, Oracle and/or its affiliates / Public

Looking at the Active Session History, we see a common picture caused by too large session pools:

Figure 9. ASH wait event graphs; timeout with increased session pools.



While we previously had the expected wait events (log file sync and resource manager waits), we now see a complex mix of wait events that are very hard to diagnose. Even in the quiet period (emulated night time) at the start of the run, there can be unexpected wait events.

The example clearly shows that using larger session pools has a *negative* effect: despite the popular belief that a larger session pool increases availability, the opposite is the case, and the overall throughput *drops* with larger amounts of cancelled requests.[8]

Wait for session instead of time out

In the previous cases, worker threads executing the actual request were configured with a time-out when getting a session from the session pool. If there was no session available, which could be either a session returned to the pool by another worker thread or a new session created by logging on to the database, the session get times out and the request is cancelled. In a real application, that would mean giving the application user a message such as “the system is too busy right now”.

An alternative to the time-out is to always wait for a session to become available, even if that means waiting for a long time.

[8] Further investigations also show that with the larger session pool, the execution time for the 90%, 95%, etc. percentiles of the emulated transactions are *higher* than they were with the modest session pool.

Copyright © 2026, Oracle and/or its affiliates / Public

We have repeated the above tests, both with the modest session pool (with a maximum pool size of 10 per process) and the large maximum session pool. The results for the modest session pool are shown in these graphs:

Figure 10. 24h simulation; wait with modest session pool.

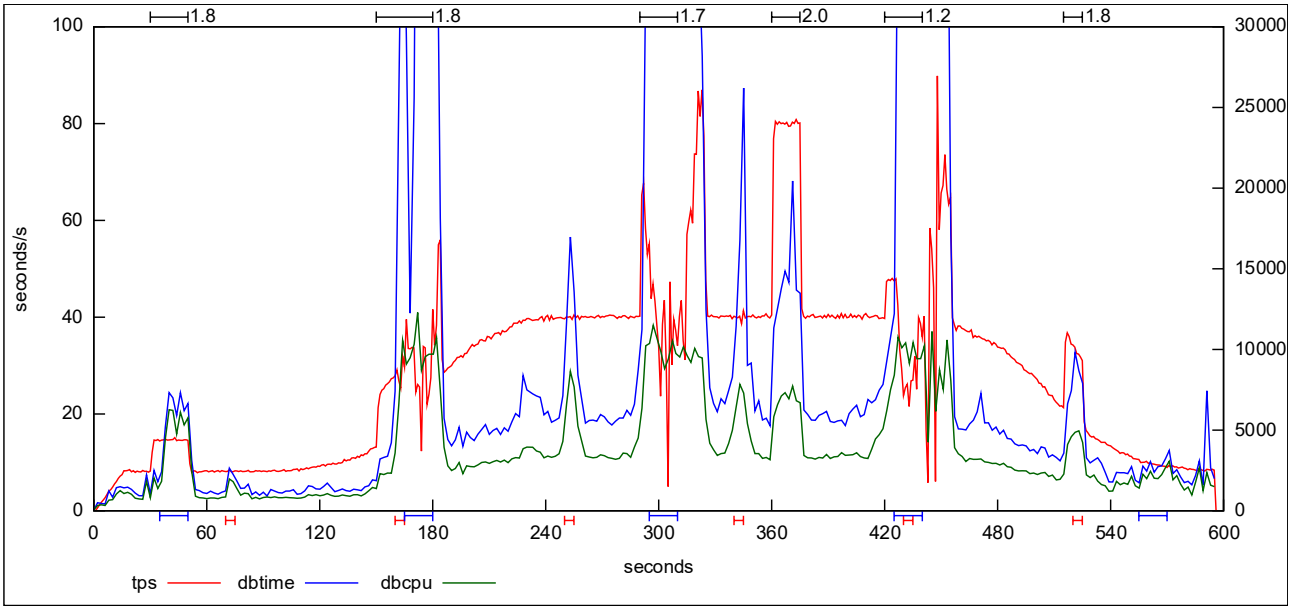
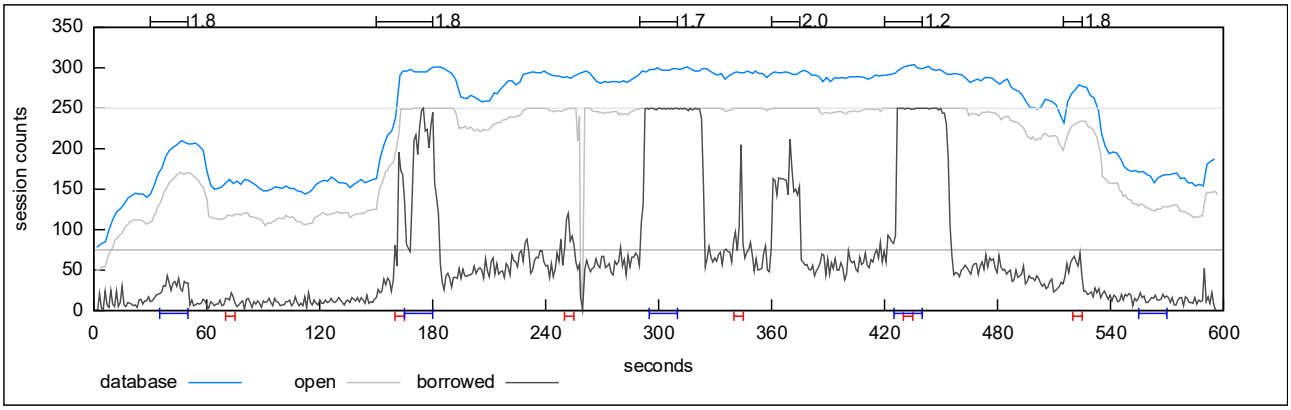


Figure 11. Sessions in 24h simulation; wait with modest session pool.

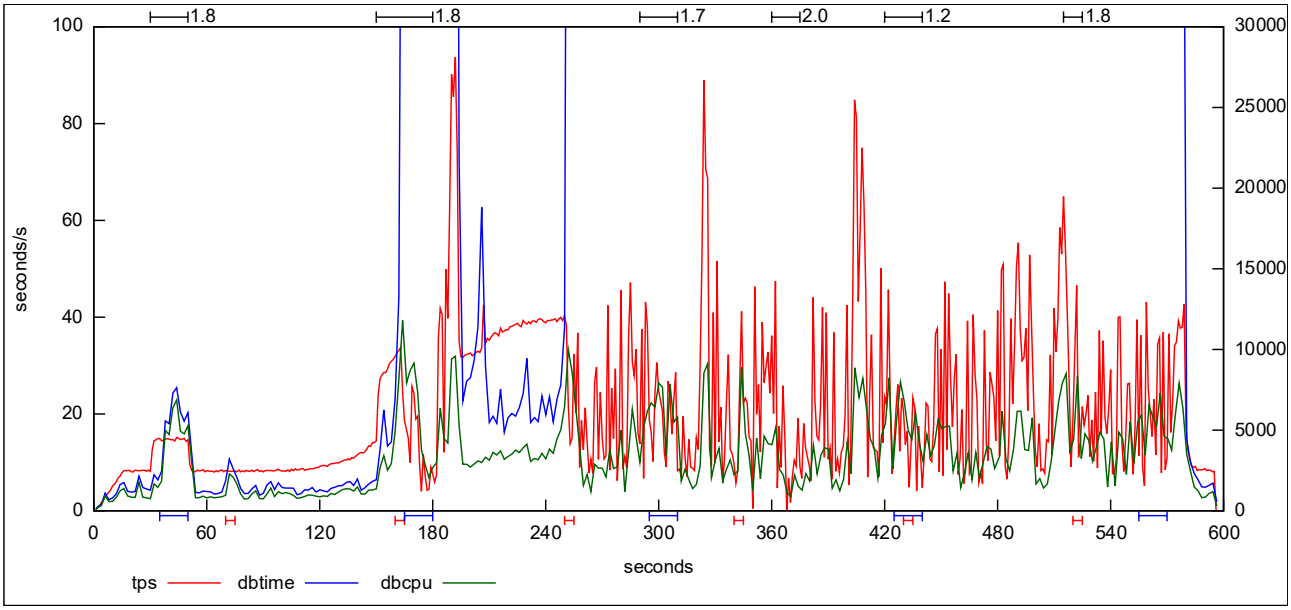


In this first case, with the modest session pool, after the injected fault is complete, there is a large backlog of requests that need to be processed, so the period with high database activity extends beyond the actual fault. Seen from the application side, the percentiles (90%, 95%, etc) are at much higher total execution times. So while no requests suffer from the “system too busy” message, there are requests that suffer from long execution times. The actual wait event seen in the Active Session History is, as in the previous test with modest session pool, primarily dominated by resource manager.

Copyright © 2026, Oracle and/or its affiliates / Public

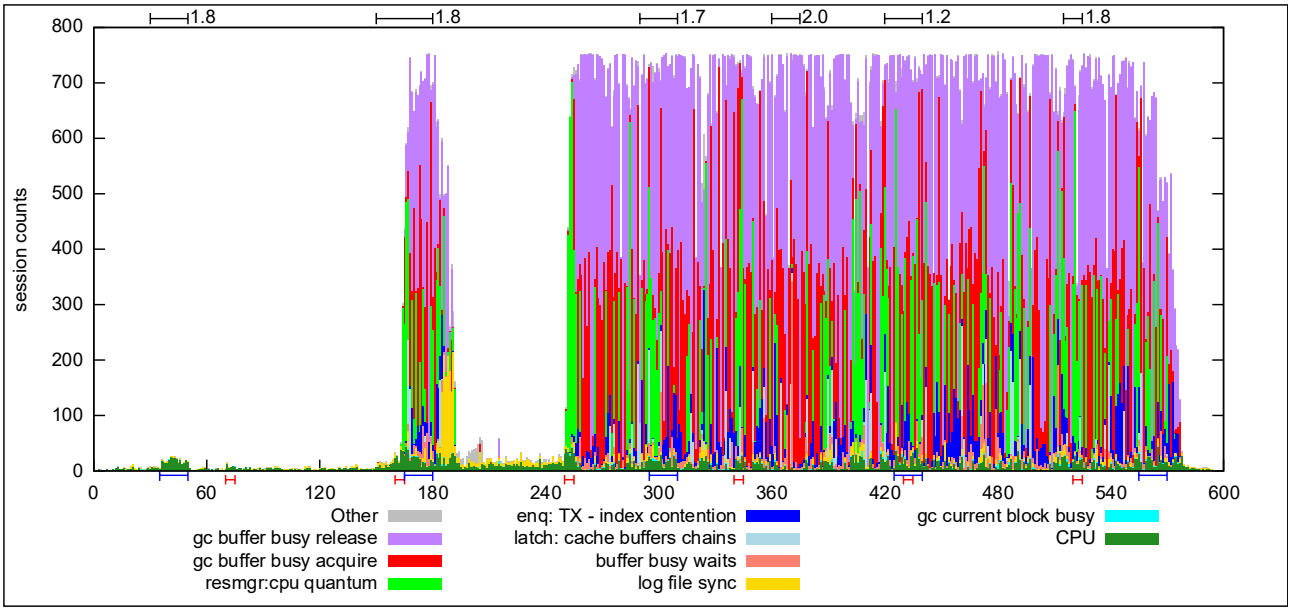
Let us finally repeat this without the time-out and with the large session pool (with a maximum pool size of 30 per process), which is shown in these graphs:

Figure 12. 24h simulation; wait with large session pool.



It becomes very obvious that the database is driven completely beyond its safe operating limits, and that that situation is present for a very long time. This behavior comes from the fact that all entries in the (too) large session pool are constantly busy serving requests. If we look at the wait events in Active Session History, we see:

Figure 13. ASH wait event graphs; wait with large session pool.



As in the case with time out and the large session pool, the database behavior is completely chaotic and unpredictable, and this situation continues for a very long time. If this had been a production system, it would have been almost completely inaccessible and would have been extremely difficult to diagnose.

Copyright © 2026, Oracle and/or its affiliates / Public

Conclusions

As discussed in Part Two, there is no single answer to the correct configuration of session pools. It is typically argued that the maximum size of the pool must be large enough to always have sessions available when requests arrive. However, as the examples clearly demonstrate, pools must be configured *modestly* to ensure the database is operating safely. Too-large pools do lead to lower throughput and higher wait times, and they further greatly increase the risk of unpredictable behavior.

We have also seen that the analysis, and therefore the ability to correctly configure session pools, *requires* the availably of metrics—in particular, metrics about the number of borrowed and open sessions in the pools.

It is also important to understand that pool sizing, based on these metrics, can only be done at full-scale workloads. Without understanding the application, the database capabilities and resources, and the workload (including expected variation over time), it is not possible to properly set the pool size parameters.

Copyright © 2026, Oracle and/or its affiliates / Public

Part Four – Statement Cache Sizing

One of the many characteristics of Online Transaction Processing is that the same SQL statements are being executed repeatedly and frequently, often hundreds of times per second. This frequently executed SQL is using placeholders, also known as bind variables, that take actual values provided by the application for each single execution. As a simplified example, the SQL:

```
insert into orders(id, custid, status) values (:id, :cust, :stat);
```

inserts the actual values of the id, customer id and status columns being provided at runtime by the application into a table of orders. To perform this, the database server roughly does these three steps[9]:

1. The text of the SQL is parsed, the presence of objects is verified, and an execution plan is generated
2. Access rights are verified.
3. The SQL actually runs, in that the actual insert of the order is performed with actual values for the three binds coming from variables in the application.

The two first steps are often referred to as *hard parse* and *soft parse,* and both take time to perform. In OLTP systems, where the same SQL is repeatedly run, these first two steps should only be done once, and only the last step is repeatedly performed. Most Application Programming Interfaces, such as JDBC or Oracle Call Interface, have a mechanism to control this based on a cache of parsed SQL statements that are ready for processing. The size of this cache, i.e. the count of SQL statements that are ready for processing, is a configuration parameter with typical values in the very rough range from 10 to 100. As just one example, the default cache in Oracle Call Interface is 20 (see [4]).

The actual setting of the statement cache size is a tradeoff between these two aspects:

1. With higher values, more statements are parsed only once, after which they are ready for processing, and overall database time is lower as fewer parses are done. However, the cache requires memory in both the application and the database.
2. With lower values, more statements are parsed before processing, and the overall database time is higher. However, less memory may be used on both application- and database-side as statements are unlikely to be kept unnecessarily open.

Let us take a look at sample code 1 shown previously in the beginning of Part Two. After `user_interface()`, it is "known" which SQL to actually run for that particular user interface, i.e. a particular business transaction, and the SQL processed while the code has borrowed a session from the session pool. At a later point in time, that same session may be borrowed to run some other business transaction, causing some other set of SQL statements to be processed. As a result, the *same* database session will, over time, run the necessary SQL of several different business transactions. As the statement cache is associated with each individual session in the session pool, the real question is how large the statement cache should be.

At one extreme, it should be so large that it can hold the total number of different SQL in all business transactions, and at the other extreme, there should be no statement cache at all. The first extreme results in the lowest possible number of parses of SQL statements, since each session in the pool only ever needs to parse each SQL statement once. The second extreme implies a parse is necessary for every single SQL execution. The first extreme has a very high memory demand, and the second causes a high overhead in parsing SQL. So, where is the balance?

[9] The full details of this are more complex and not discussed further here.

Copyright © 2026, Oracle and/or its affiliates / Public

As with sizing the session pool, there is no single correct answer. However, an *AWR report* can give a good indication. In the Load Profile, a few figures are relevant:

Table 4. AWR report statistics.

Load Profile Figure	Per Second
Parses (SQL):	59.4
Hard parses (SQL):	0.3
Executes (SQL):	4,843.9

In this case, there are 59.4 parses, 0.3 of which are hard parses, per second, and there are 4843.9 executes per second, which effectively means a parse to execute rate of about 1.2%, which is considered fully acceptable. The AWR report also has a section showing parse counts per SQL, and an extract of this from the same AWR shows:

Table 5. AWR report showing parse counts per SQL.

Parse Calls	Executions	% Total Parses	SQL Id	SQL Module
510	1,674	3.19	**1m0ku2f7pr6sz**	rwloadsim23
490	2,225	3.06	**8cwr9vd6bssf9**	rwloadsim23
479	42,499	2.99	**64jd66kntmv6d**	rwloadsim23
479	42,499	2.99	**83frmyfzcjqu3**	rwloadsim23
444	175,408	2.77	**2bw8nk7t9p5x9**	rwloadsim23
444	175,416	2.77	**5c5k87z0sdrmk**	rwloadsim23
444	159,615	2.77	**c40cqmgc4nszv**	rwloadsim23
436	72,019	2.72	**cyjqk0wv9y4u0**	rwloadsim23
436	71,971	2.72	**frrqa9hp1dcq6**	rwloadsim23

The values worth noting here are that very frequently run SQL statement, like the ones with SQL Id 2bw8nk7t9p5x9 and 5c5k87z0sdrmk, have a very low parse to execute ratio, while other less-frequently executed SQL statements have proportionally higher parse counts. Results like this are exactly the benefits of a properly sized statement cache.

Copyright © 2026, Oracle and/or its affiliates / Public

In another execution of the same application with a too-small value for the statement cache size, the same AWR extracts are:

Table 6. AWR report statistics for small statement cache size.

Load Profile Figure	Per Second
Parses (SQL):	766.4
Hard parses (SQL):	0.7
Executes (SQL):	4,993.7

Table 7. AWR report showing parse counts per SQL for small statement cache size.

Parse Calls	Executions	% Total Parses	SQL Id	SQL Module
16,214	55,058	8.15	394rhcktypag7	rwloadsim23
16,197	55,417	8.14	b2cuhk2ag2wtm	rwloadsim23
16,197	166,098	8.14	gu169nhtf69z0	rwloadsim23
16,195	55,414	8.14	4wsgmyph6a90a	rwloadsim23
12,892	66,944	6.48	frrqa9hp1dcq6	rwloadsim23
10,027	66,984	5.04	cyjqk0wv9y4u0	rwloadsim23
4,868	162,586	2.45	5c5k87z0sdrmk	rwloadsim23

In this case, the overall parse to execute ratio is around 15% and the parse to execute ratio for individual SQL is also much higher than with the properly-sized statement cache.

The proper size of the statement cache is, by nature, variable. It may even vary with the same type of variations that have been discussed for session pool sizing, such as day/night or weekday/weekend. Most importantly, the application should be designed so that it can be changed by system administrators based on information from AWR reports.

Note that if an AWR report shows that the number of parses and executes is the *same* for some SQL statements, it means that the application running that SQL does not use a statement cache. For infrequently-executed SQL, this is not a problem, but frequently-executed SQL should always use a statement cache to prevent the overhead of parsing.

Copyright © 2026, Oracle and/or its affiliates / Public

Part Five – Batch Processing

This paper primarily discusses Online Transaction Processing, but some of the concepts discussed may apply to batch processing as well. The major differences between the two are:

- Each request in a batch-oriented system is typically much longer than in an OLTP system. While an OLTP request typically lasts milliseconds, a batch request can last seconds, minutes, or even hours.
- The arrival of OLTP requests is completely random, while batch requests are planned for certain times of the day and very often planned with a reasonable concurrency using batch queues.
- The primary requirement for batch processing is that it must complete at a certain time and that it is fully accepted and expected that a request may queue for a long time before it is serviced.

To compare to the previous OLTP discussion, batch processing is almost solely about transaction rate ("getting the batch done"), and the response time is largely irrelevant, or at least of a very different nature.

While a batch request is being processed, potentially taking minutes or hours, there is no wait for user input, so there is no need for session pools. The pseudo code therefore changes:

Sample Code 2

```
loop ...
  get_request_from_queue();
  application_logic0;
  database_logon();
    loop ...
      sql1;
      application_logic1;
      sql2;
      application_logic2;
      sql3;
      application_logic3;
      commit;
    end loop;
  database_logoff();
end loop;
```

Compared to OLTP, each batch request is much longer (minutes, hours) and each batch request therefore establishes its own database session that is held until the request is complete. If there are always requests in the batch queue (that is, if the `get_request_from_queue()` call above never waits), the situation can be somewhat comparable to the OLTP case with all sessions in the pool being constantly busy. The requirement to keep the database within its safe operating limits is as important for batch processing as it is for OLTP. As a result, we need to put a limit on the number of concurrent executions of code like sample code 2. Note that possible database-side parallel execution of the individual SQL highly influences this.[10]

Also note that the foreground idle time can still be very relevant, as there is still networking between the application and the database servers, and there is still application logic being processed on the former.

Batch processing, like OLTP, can repeat the same SQL statements with only changes to values of placeholders. In this case, the use of the statement cache is as relevant for batch processing as it is for OLTP. In other cases, batch processing is of a nature where the same SQL is rarely repeatedly processed, which has two important implications:

- The use of a statement cache is irrelevant as the purpose of the statement cache is to improve performance of repeatedly-executed SQL by only doing the parse call once.

[10] In autonomous databases, this is reflected in the resource management setting for the low, medium, and high services.

Copyright © 2026, Oracle and/or its affiliates / Public

- Because of that, non-repeated SQL in batch processes should *not* use placeholders (bind variables), but alternatively have literals in the SQL statement, as that allows for better optimization of the SQL statement, eventually giving shorter processing times.

Copyright © 2026, Oracle and/or its affiliates / Public

Part Six – Summary

The Oracle database is at the core of a vast variation of application environments, and supports a vast variation of business processes. As the database is at the "bottom" of an often-complex stack, availability of the database must be a primary concern, and must at all means be protected against failures, including failures that are the result of overloading. The database—like any other resources—is not unlimited.

A very efficient way to ensure the database is not overloaded is to never establish any more sessions in the database than those sessions that can all be active at the same time. As mentioned in [1], having applications use a limited pool of sessions is an efficient means to achieve this goal. However, due to complexity, variation over time, and variation in workload, just having a single or a few such pools is, for all practical purposes, only achievable in rare and unrealistic cases. Session pool sizing always includes tradeoffs: If pools are too large or there are too many, there is a risk of the database overloading. If pools are too small or there are too few, the database may not utilize all available resources, making users wait even when the database could have handled the incoming requests.

In this paper, we discussed the typical configuration options for session pools, and showed with actual examples that the frequently argued statement that "the pool must be large enough to always handle requests" may very well have a negative effect on both throughput and response time. We also demonstrated that having the system configured to give some users a "system is too busy right now" message is clearly favorable to the alternative of letting all users send requests, as that approach has a high risk of impacting all users.

We also showed the importance of having actual metrics about pool usage available.

Copyright © 2026, Oracle and/or its affiliates / Public

Appendix A – References

Table 8. References.

Reference	Topic	URL
1	Application Programming Using Pooling	https://download.oracle.com/ocomdocs/global/Application_Programming_Using_Pooling.pdf
2	Connection Pool Sizing and SmartDB	https://www.youtube.com/watch?v=eiydITTdDAQ
3	The RWP*Load Simulator (with specific attention to the OLTP.md section)	https://oracle.github.io/rwloadsim/
4	Performance Topics in Oracle Call Interface Developer's Guide (with specific attention to the section on statement caching)	https://docs.oracle.com/en/database/oracle/oracle-database/26/lnoci/performance-topics.html
5	Session Pooling and Connection Pooling in OCI in Oracle Call Interface Developer's Guide (with specific attention to the section on session pooling)	https://docs.oracle.com/en/database/oracle/oracle-database/26/lnoci/session-and-connection-pooling.html
6	Statement and Result Set Caching in JDBC Developer's Guide (with specific attention to the section on statement caching)	https://docs.oracle.com/en/database/oracle/oracle-database/26/jjdbc/statement-and-resultset-caching.html
7	Introduction to UCP in Universal Connection Pool Developer's Guide (with specific attention to this paper's use of the term "session pool")	https://docs.oracle.com/en/database/oracle/oracle-database/26/jjucp/intro.html

Copyright © 2026, Oracle and/or its affiliates / Public

Primary author: Bjørn Kisbye Engsig
Contributing authors: Toon Koppelars, Mike Hallas, Sharad Chandran R

Connect with us

Call +**1.800.ORACLE1** or visit **oracle.com**. Outside North America, find your local office at: **oracle.com/contact**.

blogs.oracle.com facebook.com/oracle twitter.com/oracle

Copyright © 2026, Oracle and/or its affiliates. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Copyright © 2026, Oracle and/or its affiliates / Public