2

Connection Strategies for Database Applications

A **database connection** is a physical communication pathway between a client process and a database instance. A **database session** is a logical entity in the database instance memory that represents the state of a current user login to a database. A session lasts from the time the user is authenticated by the database until the time the user disconnects or exits the database application. A single connection can have 0, 1, or more sessions established on it.

Most OLTP performance problems that the Oracle Real-World Performance group investigates relate to the application connection strategy. For this reason, designing a sound connection strategy is crucial for application development, especially in enterprise environments that must scale to meet increasing demand.

Topics:

2.1 Design Guidelines for Connection Pools

A **connection pool** is a cache of connections to an Oracle database that is usable by an application.

At run time, the application requests a connection from the pool. If the pool contains a connection that can satisfy the request, then it returns the connection to the application. The application uses the connection to perform work on the database, and then returns the connection to the pool. The released connection is then available for the next connection request.

In a **static connection pool**, the pool contains a fixed number of connections and cannot create more to meet demand. Thus, if the pool cannot find an idle connection to satisfy a new application request, then the request is queued or an error is returned. In a **dynamic connection pool**, however, the pool creates a new connection, and then returns it to the application. In theory, dynamic connection pools enable the number of connections in the pool to increase and decrease, thereby conserving system resources that are otherwise lost on maintaining unnecessary connections. However, in practice, a dynamic connection pool strategy allows for potential connection storms and over-subscription problems.

2.1.1 Connection Storms

A **connection storm** is a race condition in which application servers initiate an increasing number of connection requests, but the database server CPU is unable to schedule them immediately, which causes the application servers to create more connections.

During a connection storm, the number of database connections can soar from hundreds to thousands in less than a minute.

Dynamic connection pools are particularly prone to connection storms. As the number of connection requests increases, the database server becomes oversubscribed relative to the number of CPU cores. At any given time, only one process can run on a CPU core. Thus, if 32 cores exist on the server, then only 32 processes can be doing work at one time. If the application server creates hundreds or thousands of connections, then the CPU becomes busy trying to keep up with the number of processes fighting for time on the system.

Inside the database, wait activity increases as the number of active sessions increase. You can observe this activity by looking at the wait events in ASH and AWR reports. Typical wait events include latches on enqueues, row cache objects, latch free, enq: TX - index contention, and buffer busy waits. As the wait events increase, the transaction throughput decreases because sessions are unable to perform work. Because the server computer is oversubscribed, monitoring tool processes must fight for time on the CPU. In the most extreme case, using the keyboard becomes impossible, making debugging difficult.

> **Video:**
>
> RWP #13: Large Dynamic Connection Pools - Part 1

2.1.2 Guideline for Preventing Connection Storms: Use Static Pools

The Oracle Real-World Performance group recommends that you use static connection pools rather than dynamic connection pools.

Over years of diagnosing connection storms, the Oracle Real-World Performance group has discovered that dynamic connection pools often use too many processes for the necessary workload. A prevalent myth is that a dynamic connection pool creates connections as required and reduces them when they are not needed. In reality, when the connection pool is exhausted, application servers enable the size of the pool of database connections to increase rapidly. The number of sessions increases with little load on the system, leading to a performance problem when all the sessions become active.

Because dynamic connection pools can destabilize the system quickly, the Oracle Real-World Performance group recommends that you use static rather than dynamic connection pools.

Reducing the number of connections reduces the stress on the CPU, which leads to faster response time and higher throughput. This result may seem paradoxical. Performance improves for the following reasons:

- Fewer sessions means fewer contention-related wait events inside the database. After reducing connections, the CPU that formerly consumed cycles on latches and arbitrating contention can spend more time processing database transactions.
- As the number of connections decreases, connections can stay scheduled on the CPU for longer. As a result, all memory pages associated with these processes stay resident in the CPU cache. They become increasingly efficient at scheduling, and stall less in memory

As a rule of thumb, the Oracle Real-World Performance group recommends a 90/10 ratio of %user to %system CPU utilization, and an average of no more than 10 processes per CPU core on the database server. The number of connections should be based on the number of CPU cores and not the number of CPU core threads. For example, suppose a server has 2 CPUs and each CPU has 18 cores. Each CPU core has 2 threads. Based on the Oracle Real-Wold Performance group guidelines, the application can have between 36 and 360 connections to the database instance.

> **Video:**
>
> RWP #14: Large Dynamic Connection Pools - Part 2

2.2 Design Guideline for Login Strategy

A problem facing all database developers is how and when the application logs in to the database to initiate transactions.

In a suboptimal design, the database application performs the following steps for every SQL request:

1. Log in to the database.
2. Issue a SQL request, such as an `INSERT` or `UPDATE` statement.
3. Log out of the database.

Applications that use a login/logout strategy may meet functional requirements. Also, they may perform well when the number of transactions per second is low. However, logging in and out of the database is an extremely resource-intensive operation. The Oracle Real-World Performance group has found that applications that use such algorithms do not scale well and can cause severe performance problems, especially when used with dynamic connection pools. A login/logout strategy usually uses no connection pool.

If an application uses a login/logout design, and if the DBAs and developers do not realize the source of the problem, then the first symptoms may be low database throughput and erratic, excessively high response times. A diagnostic investigation of the database may show that relatively few sessions are active, while resource contention is low.

The clue to the suboptimal performance is that the number of logins per second is close to the number of transactions per second. When a login/loutout per transaction stategy is used, the database instance and operating system perform a lot of work behind the scenes to create the new process, database connection, and associated memory area. Many of these steps are serialized, leading to low CPU utilization combine with low transaction througput.

For the preceding reasons, the Oracle Real-World Performance group strongly recommends against a login/logout design for any application that must scale to support a high number of transactions.

> **Video:**
>
> RWP #2 Bad Performance with Cursors and Logons

2.3 Design Guideline for Preventing Programmatic Session Leaks

A **session leak** occurs when a program loses a connection, but its session remains active in the database instance. A leaked session is programmatically lost to the application.

An optimally designed application prevents session leaks. Typically, session leaks occur because of exceptions caught by the application. If the application does not handle the

exception correctly, then it may terminate the connection without executing a commit or rollback, thus leaking the session.

Session leaks can cause severe problems for database performance and data integrity. Typically, the problems take the following forms:

- Drained connection pools
- Lock leaks
- Logical corruption

2.3.1 Drained Connection Pools

Design flaws can cause connection pools to drain.

For example, assume that an application design flaw causes it to leak sessions consistently. Even if the leak rate is low, a dynamic connection pool leads to an ever-increasing number of sessions become programmatically impossible to use.

The effect is to reduce the usable connection pool and prevent the remaining connections from keeping up with the workload. The number of unusable sessions climbs until there are no usable connections left in the pool.

2.3.2 Checking for Session Leaks

Session leaks occur due to issues in the application or application server and cannot be fixed from the database alone. The issue needs to be addressed in the application or application server. An easy way to check for session leaks is by modifying the connection pool to use one connection to the database and test the application. Testing with one connection makes it is easier to find the root cause of the problem in the application.

2.3.3 Lock Leaks

A **lock leak** is typically a side-effect of a session leak.

For example, a leaked session that was in the middle of a batch update may hold TX locks on multiple rows in a table. If a leaked session is holding a lock, then sessions that want to acquire the lock form a queue behind the leaked session.

The program that is holding the lock is waiting for interaction from the client to release that lock, but because the connection is lost programmatically, the message will not be sent. Consequently, the database cannot commit or roll back any transaction active in the session.

2.3.4 Logical Corruption

A leaked session can contain uncommitted changes to the database. For example, a transaction is partway through its work when the database connection is unexpectedly released.

This situation can lead to the following problems:

- The application reports an error to the UI. In this case, customers may complain that they have lost work, for example, a business order or a flight schedule
- The UI receive a commit message even though no commit or rollback has occurred. This is the worst case, because a subsequent transaction might commit both its own work and the

half of the transaction from the leaked session. In this case, the database is logically corrupted.

2.4 Using Runtime Connection Load Balancing

Topics:

2.4.1 About Runtime Connection Load Balancing

Oracle Real Application Clusters (Oracle RAC) is a database option in which a single database is hosted by multiple instances on multiple nodes. The Oracle RAC shared disk method of clustering databases increases scalability. The nodes can easily be added or freed to meet current needs and improve availability, because if one node fails, another can assume its workload. Oracle RAC adds high availability and failover capacity to the database, because all instances have access to the whole database.

Work requests are balanced at both connect time (**connect time load balancing**, provided by Oracle Net Services) and runtime (**runtime connection load balancing**). For Oracle RAC environments, session pools use service metrics received from the Oracle RAC load balancing advisory through Fast Application Notification (FAN) events to balance application session requests. The work requests coming into the session pool can be distributed across the instances of Oracle RAC offering a service, using the current service performance.

Connect time load balancing occurs when an application creates a session. Pooled sessions must be well distributed across Oracle RAC instances when the sessions are created to ensure that sessions on each instance can execute work.

Runtime connection load balancing occurs when an application selects a session from an existing session pool (and thus is a very frequent activity). Runtime connection load balancing routes work requests to sessions in a session pool that best serve the work. For session pools that support services at only one instance, the first available session in the pool is adequate. When the pool supports services that span multiple instances, the work must be distributed across instances so that the instances that are providing better service or have greater capacity get more work requests.

OCI, OCCI, JDBC, and ODP.NET client applications all support runtime connection load balancing.

See Also:

- `cdemosp.c` in the directory `demo`
- Using Database Resident Connection Pool
- *Oracle Call Interface Programmer's Guide*
- *Oracle C++ Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide*
- *Oracle Universal Connection Pool for JDBC Java API Reference*
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows*

2.4.2 Enabling and Disabling Runtime Connection Load Balancing

Enabling and disabling runtime connection load balancing on the client depends on the client environment.

Topics:

- OCI
- OCCI
- JDBC
- ODP.NET

2.4.2.1 OCI

For an OCI client application, runtime connection load balancing is enabled by default in an Oracle Database 11*g* Release 1 (11.1) or later client communicating with a server of Oracle Database 10*g* Release 2 (10.2) or later when you perform the following operations to ensure that your application receives service metrics based on service time:

- Link the application with the threads library.
- Create the OCI environment in `OCI_EVENTS` and `OCI_THREADED` modes.
- Configure the load balancing advisory goal and the connection load balancing goal for a service that is used by the session pool.

To disable runtime connection load balancing for an OCI client, set the `mode` parameter to `OCI_SPC_NO_RLB` when calling `OCISessionPoolCreate()`.

FAN HA (FCF) for OCI requires `AQ_HA_NOTIFICATIONS` for the service to be `TRUE`.

See Also:

Oracle Call Interface Programmer's Guide for information about `OCISessionPoolCreate()`

2.4.2.2 OCCI

For an OCCI client application, runtime connection load balancing is enabled by default in an Oracle Database 11*g* Release 1 (11.1) or later client communicating with a server of Oracle Database 10*g* Release 2 (10.2) or later when you perform the following operations:

- Link the application with the threads library.
- Create the OCCI environment in `EVENTS` and `THREADED_MUTEXED` modes.
- Configure the load balancing advisory goal and the connection load balancing goal for a service that is used by the session pool.

To disable runtime connection load balancing for an OCCI client, use the `NO_RLB` option for the `PoolType` attribute of the `StatelessConnectionPool Class`.

FAN HA (FCF) for OCCI requires `AQ_HA_NOTIFICATIONS` for the service to be `TRUE`.

> **See Also:**
>
> *Oracle C++ Call Interface Programmer's Guide* for more information about runtime load balancing using the OCCI interface

2.4.2.3 JDBC

In the JDBC environment, runtime connection load balancing is enabled by default in an Oracle Database 11*g* Release 1 (11.1) or later client communicating with a server of Oracle Database 10*g* Release 2 (10.2) or later when Fast Connection Failover (FCF) is enabled.

In the JDBC environment, runtime connection load balancing relies on the Oracle Notification Service (ONS) infrastructure, which uses the same out-of-band ONS event mechanism used by FCF processing. No additional setup or configuration of ONS is required to benefit from runtime connection load balancing.

To disable runtime connection load balancing in the JDBC environment, call `setFastConnectionFailoverEnabled()` with a value of false.

> **See Also:**
>
> *Oracle Database JDBC Developer's Guide* for more information about runtime load balancing using the JDBC interface

2.4.2.4 ODP.NET

In an ODP.NET client application, runtime connection load balancing is disabled by default. To enable runtime connection load balancing, include "`Load Balancing=true`" in the connection string and make sure "`Pooling=true`" (default).

FAN HA (FCF) for ODP.NET requires `AQ_HA_NOTIFICATIONS` for the service to be `TRUE`.

> **See Also:**
>
> *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows* for more information about runtime load balancing

2.4.3 Receiving Load Balancing Advisory FAN Events

Your application can receive load balancing advisory FAN events only if all of these requirements are met:

- Oracle RAC environment with Oracle Clusterware is set up and enabled.
- The server is configured to issue event notifications.
- The application is linked with the threads library.
- The OCI environment is created in `OCI_EVENTS` and `OCI_THREADED` modes.
- The OCCI environment is created in `THREADED_MUTEXED` and `EVENTS` modes.
- You configured or modified the Oracle RAC environment using the `DBMS_SERVICE` package.

  You must modify the service to set up its goal and the connection load balancing goal as follows:

  ```
  EXEC DBMS_SERVICE.MODIFY_SERVICE("myService",
       DBMS_SERVICE.GOAL_SERVICE_TIME,
       clb_goal => DBMS_SERVICE.CLB_GOAL_SHORT);
  ```

  The constant `GOAL_SERVICE_TIME` specifies that Load Balancing Advisory is based on elapsed time for work done in the service plus bandwidth available to the service.

  The constant `CLB_GOAL_SHORT` specifies that connection load balancing uses Load Balancing Advisory, when Load Balancing Advisory is enabled.You can set the connection balancing goal to `CLB_GOAL_LONG`. However, `CLB_GOAL_LONG` is typically useful for closed workloads (that is, when the rate of completing work is equal to the rate of starting new work).

> **See Also:**
>
> - *Oracle Real Application Clusters Administration and Deployment Guide* for information about enabling OCI clients to receive FAN events
> - *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_SERVICE`
> - *Oracle Call Interface Programmer's Guide* for information about `OCISessionPoolCreate()`
> - *Oracle Database JDBC Developer's Guide* for more information about runtime load balancing using the JDBC interface