3
Performance and Scalability

Explains techniques for designing performance and scalability into the database and database applications.

Topics:

- Performance Strategies
- Tools for Performance
- Monitoring Database Performance
- Testing for Performance
- Using Client Result Cache
- Statement Caching
- OCI Client Statement Cache Auto-Tuning
- Client-Side Deployment Parameters
- Using Query Change Notification
- Using Database Resident Connection Pool
- Memoptimize Pool
- Oracle RAC Sharding

3.1 Performance Strategies

Topics:

- Designing Your Data Model to Perform Well
- Setting Performance Goals (Metrics)
- Benchmarking Your Application

3.1.1 Designing Your Data Model to Perform Well

This topic briefly describes the important concepts of data modeling. Entire books about this subject are available; refer to them for details and further guidance.

Design your data model for optimal performance of the most important and frequent queries, following this basic procedure:

1. Analyze the Data Requirements of the Application
2. Create the Database Design for the Application
3. Implement the Database Application
4. Maintain the Database and Database Application

3.1.1.1 Analyze the Data Requirements of the Application

Analyze the data requirements of your application by following this basic procedure:

1. Collect data.

   Interview people to learn about the business, the nature of the application, who uses information and how, and the expectations of end users. Collect business documents—personnel forms, invoice forms, order forms, and so on—to learn how the business uses information.

2. Analyze the collected data.

   This bottom-up process includes normalization of the data, entity-relationship modeling, and transaction analysis.

3. Do a functional analysis of the data.

   The end result of this top-down process is a data flow diagram that identifies the main process blocks and how data flows into and out of them over its life time.

3.1.1.2 Create the Database Design for the Application

To create the database design, you must first create the logical design, and then translate this logical design into a physical design.

Topics:

- Create the Logical Design
- Create the Physical Design

See Also:

Oracle Database Performance Tuning Guide for more information about designing and developing for performance

3.1.1.2.1 Create the Logical Design

The logical design is a graphical representation of the database. The logical design models both relationships between database objects and transaction activity of the application. Effective logical design considers the requirements of different users who must own, access, and update data.

To model relationships between database objects:

1. Translate the data requirements into data items (columns).
2. Group related columns into tables.
3. Map relationships among columns and tables, determining primary and foreign key attributes for each table.
4. Normalize the tables to minimize redundancy and dependency.

To model transaction activity:

- Know the most common transactions and those that users consider most important.

- Trace transaction paths through the logical model.
- Prototype transactions in SQL and develop a volume table that indicates the size of your database.
- Determine which tables are accessed by which users in which sequences, which transactions read data, and which transactions write data.
- Determine whether the application mostly reads data or mostly writes data.

3.1.1.2.2 Create the Physical Design

The physical design is the implementation of the logical design on the physical database.

Because the logical design integrates the information about tables and columns, the relationships between and among tables, and all known transaction activity, you know how the database stores data in tables and creates other structures, such as indexes.

Using this knowledge, create scripts that use SQL data definition language (DDL) to create the schema definition, define the database objects in their required sequence, define the storage requirements to specification, and so on.

3.1.1.3 Implement the Database Application

Implement the database application by following this basic procedure:

1. Implement the application in a test environment.

   In a test environment that is as similar as possible to the production environment, run the scripts that implement the physical database design. Load the data into the physical schema. Select the programming language in which to develop the application, develop the user interface, create and test the transactions, and so on.
2. Ensure that the application runs to specification.

   Ensure that all components are exercised, the application is fully operational, and the database features that the application uses are optimally configured.
3. Run benchmark tests on the application.

   Benchmark tests determine whether the application performs as expected under various workloads (including peak activity) with simulated real-time operations, such as adding data and users. Ensure that the application scales well.

   If the application does not meet the benchmarks, tune your SQL statements to perform optimally, first with no workload and then with increasing workloads.
4. Implement the application in the production environment.

See Also:

- *Oracle Database SQL Tuning Guide* for more information about SQL tuning
- *Oracle Database 2 Day + Performance Tuning Guide* for more information about SQL tuning
- *Oracle Database Performance Tuning Guide* for more information about workload testing, modeling, and implementation
- Tools for Performance for information about tools for testing performance
- Benchmarking Your Application for information about benchmarking your application
- *Oracle Database Performance Tuning Guide* for more information about deploying new applications

3.1.1.4 Maintain the Database and Database Application

Maintaining the database, the database application, and the operating system are on-going tasks for the database administrator, the application developer, and the system administrator, respectively. The resources that the business allocates to maintenance depend on the importance of the database and the database application, its growth potential, the need to accommodate more users, and so on.

If you are responsible for maintenance, you must periodically monitor the system, schedule maintenance periods, and inform users of upcoming maintenance periods. If maintenance periods require down time, schedule them for periods with little or no database activity.

Application maintenance includes fixing bugs, applying patches, and releasing upgrades. Test maintenance work in a test environment to catch and resolve any before implemented it on production systems.

3.1.2 Setting Performance Goals (Metrics)

Start your application development project by setting performance goals (metrics), including:

- Expected number of application users
- Expected number of transactions per second at peak load times
- Expected query response times at peak load times
- Expected number of records for each table per unit of time (such as one day, one month, or one year)

Use these metrics to create benchmark tests.

3.1.3 Benchmarking Your Application

Benchmarks are tests that measure aspects of application performance. Benchmark results either validate application design or raise issues that you can resolve before putting the application into production.

Usually, you first run benchmarks on an isolated single-user system to minimize interference from other factors. Results from such benchmarks provide a performance baseline for the application. For meaningful benchmark results, you must test the application in the environment where you expect it to run.

You can create small benchmarks that measure performance of the most important transactions, compare different solutions to performance problems, and help resolve design issues that could affect performance.

You must develop much larger, more complex benchmarks to measure application performance during peak user loads, peak transaction loads, or both. Such benchmarks are especially important if you expect the user or transaction load to increase over time. You must budget and plan for such benchmarks.

After the application is in production, run benchmarks regularly in the production environment and store their results in a database table. After each benchmark run, compare the previous and new records for transactions that cannot afford performance degradation. Using this method, you isolate issues as they arise. If you wait until users complain about performance, you might be unable to determine when the problem started.

See Also:

Oracle Database Performance Tuning Guide for more information about benchmarking applications

3.2 Tools for Performance

Several tools report runtime performance information about your application.

Topics:

- DBMS_APPLICATION_INFO Package
- SQL Trace Facility (SQL_TRACE)
- EXPLAIN PLAN Statement

See Also:

Oracle Database Testing Guide for more information about tools for tuning the database

3.2.1 DBMS_APPLICATION_INFO Package

Use the `DBMS_APPLICATION_INFO` package with the SQL Trace facility and Oracle Trace and to record the names of executing modules or transactions in the database. System administrators and performance tuning specialists can use this recorded information to track the performance of individual modules and for debugging. System administrators can also use this information to track resource use by module.

When you register the application with the database, its name and actions are recorded in the views `V$SESSION` and `V$SQLAREA`.

The DBMS_APPLICATION_INFO package provides subprograms that set the following columns in the V$SESSION view:

- MODULE (name of application or package)
- ACTION (name of transaction or packaged subprogram)
- CLIENT_INFO (additional information about the client application, such as initial bind variable values for the current session)

The DBMS_APPLICATION_INFO package also provides subprograms that return information from the preceding V$SESSION columns for the current session.

You can also use the DBMS_APPLICATION_INFO package to track the progress of commands that take many seconds to display results (such as those that create indexes or update many rows). The DBMS_APPLICATION_INFO package provides a subprogram that stores information about the command in the V$SESSION_LONGOPS view. The V$SESSION_LONGOPS view shows when the command started, how far it has progressed, and its estimated time to completion.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_APPLICATION_INFO package
- SQL Trace Facility (SQL_TRACE) for more information about DBMS_APPLICATION_INFO package with the SQL Trace facility
- *Oracle Database Reference* for information about the V$SESSION view
- *Oracle Database Reference* for information about the V$SESSION_LONGOPS view
- *Oracle Database Reference* for information about the V$SQLAREA view

3.2.2 SQL Trace Facility (SQL_TRACE)

Use the SQL Trace facility, SQL_TRACE, to trace all SQL statements and PL/SQL blocks that your application executes. By enabling the SQL Trace facility for a single statement or module and then disabling it after the statement or module runs, you can get trace information for only that statement or module.

For best results, use the SQL Trace facility with TKPROF and the EXPLAIN PLAN statement. The SQL Trace facility provides performance information for individual SQL statements. TKPROF formats the trace file contents in a readable file. The EXPLAIN PLAN statement shows the execution plans chosen by the optimizer and the execution plan for the specified SQL statement if it were executed in the current session.

Use the SQL Trace facility while designing your application, so that you know what you want to trace and how the application performs before putting it into production. If you wait until performance problems develop in the production environment, your application structure might make using the SQL Trace facility almost impossible.

See Also:

- EXPLAIN PLAN Statement
- *Oracle Database SQL Tuning Guide* for more information about `SQL_TRACE` and `TKPROF`

3.2.3 EXPLAIN PLAN Statement

When you run a SQL statement, the optimizer generates several possible execution plans, calculates the cost of each plan, and uses the plan with the lowest cost.

Plan cost is based on statistics such as the data distribution and storage characteristics of the tables, indexes, and partitions that the SQL statement accesses. The cost of access paths and join orders is based on estimated use of computer resources such as I/O, CPU, and memory.

When you use the statement `EXPLAIN PLAN FOR` *`statement`* before running *`statement`*, the `EXPLAIN PLAN` statement stores the execution plan for *`statement`* in a plan table. By querying the plan table, you can examine the execution plan that the optimizer chose.

The plan table presents the execution plan as a row source tree, which shows the steps that Oracle Database uses to execute the SQL statement. The row source tree shows the order in which the statement references tables, the join method for tables affected by join operations, and data operations such as filtering, sorting, and aggregation. The plan table also contains optimization information, such as the cost and cardinality of each operation, partitioning information (such as the set of accessed partitions), and parallel execution information (such as the distribution method of join inputs). This information provides valuable insight into how and why the optimizer chose the execution plan.

If you have information about the data that the optimizer does not have, you can give the optimizer a hint, which might change the execution plan. If tests show that the hint improves performance, keep the hint in your code.

Note:

You must regularly test code that contains hints, because changing database conditions and query performance enhancements in subsequent releases can significantly impact how hints affect performance.

The `EXPLAIN PLAN` statement shows only how Oracle Database would run the SQL statement when the statement was explained. If the execution environment or explain plan environment changes, the optimizer might use a different execution plan. Therefore, Oracle recommends using SQL plan management to build a SQL plan baseline, which is a set of accepted execution plans for a SQL statement.

First, use the `EXPLAIN PLAN` statement to see the statement's execution plan. Second, test the execution plan to verify that it is optimal, considering the statement's actual resource consumption. Finally, if the plan is optimal, use SQL plan management.

See Also:

- *Oracle Database SQL Tuning Guide* for more information about the query optimizer
- *Oracle Database SQL Tuning Guide* for more information about query execution plans
- *Oracle Database SQL Tuning Guide* for more information about influencing the optimizer with hints
- *Oracle Database SQL Tuning Guide* for more information about SQL plan management

3.3 Monitoring Database Performance

Oracle Database provides advisors and powerful tools to help you manage and tune your database.

Topics:

- Automatic Database Diagnostic Monitor (ADDM)
- Monitoring Real-Time Database Performance
- Responding to Performance-Related Alerts
- SQL Advisors and Memory Advisors

3.3.1 Automatic Database Diagnostic Monitor (ADDM)

Automatic Database Diagnostic Monitor (ADDM) is an advisor that analyzes data captured in Automatic Workload Repository (AWR). ADDM and AWR are part of the Oracle Diagnostic Pack.

ADDM determines where database performance problems might exist and where they do not exist, and recommends corrections for the former.

ADDM performs its analysis after each AWR snapshot and stores the results in the database. By default, the AWR snapshot interval is 1 hour and the ADDM results retention period is 8 days.

Oracle Enterprise Manager Cloud Control (Cloud Control) displays ADDM Findings on the Database home page. This AWR snapshot data confirms ADDM results, but lacks the analysis and recommendations that ADDM provides. (AWR snapshot data is similar to Statspack data that database administrators used for performance analysis.)

See Also:

Oracle Database 2 Day + Performance Tuning Guide for more information about configuring ADDM, reviewing ADDM analysis, interpreting ADDM findings, implementing ADDM recommendations, and viewing snapshot statistics using Enterprise Manager

3.3.2 Monitoring Real-Time Database Performance

Using Oracle Enterprise Manager Cloud Control (Cloud Control), you can monitor real-time database performance from the Performance page. From the Performance page, you can access pages that identify performance issues and resolve those issues without waiting for the next ADDM analysis.

See Also:

Oracle Database 2 Day + Performance Tuning Guide for more information about monitoring real-time database performance

3.3.3 Responding to Performance-Related Alerts

The Database home page in Oracle Enterprise Manager Cloud Control (Cloud Control) displays performance-related alerts generated by the database. You can improve database performance by resolving problems indicated by these alerts. Oracle Database by default enables alerts for tablespace usage, snapshot too old, recovery area low on free space, and resumable session suspended. You can view metrics and thresholds, set metric thresholds, respond to alerts, clear alerts, and set up direct alert email notification. Using this built-in alerts infrastructure allows you to be notified for these special performance-related alerts.

See Also:

- *Oracle Database Administrator's Guide* for more information about using alerts to help you monitor and tune the database and managing alerts
- *Oracle Database 2 Day + Performance Tuning Guide* for more information about monitoring performance alerts

3.3.4 SQL Advisors and Memory Advisors

Oracle Database provides SQL advisors and memory advisors that you can use to help improve database performance.

SQL advisors include SQL Tuning Advisor and SQL Access Advisor, which are part of the Oracle Database Tuning Pack.

SQL Tuning Advisor accepts one or more SQL statements as input and returns specific tuning recommendations. Each recommendation includes a rationale and expected benefit. Recommendations are based on object statistics, new index creation, SQL statement restructuring, and SQL profile creation.

SQL Access Advisor enables you to optimize data access paths of SQL queries by recommending a set of materialized views and view logs, indexes, and partitions for a given SQL workload.

Memory advisors include Memory Advisor, SGA Advisor, Shared Pool Advisor, Buffer Cache Advisor, and PGA Advisor. Memory advisors provide graphical analyses of total memory target settings, SGA and PGA target settings, and SGA component size settings. Use these analyses to tune database performance and to plan for possible situations.

See Also:

- *Oracle Database 2 Day + Performance Tuning Guide* for more information about tools for tuning the database
- *Oracle Database SQL Tuning Guide* for more information about SQL Tuning Advisor
- *Oracle Database SQL Tuning Guide* for more information about SQL Access Advisor

3.4 Testing for Performance

When testing an application for performance, follow these guidelines:

- Use Automatic Database Diagnostic Monitor (ADDM) and SQL Tuning Advisor for design validation.

  ADDM determines where database performance problems might exist and recommends corrections. For example, if ADDM finds high-load SQL statements, then you can use SQL Tuning Advisor to analyze those statements and provide tuning recommendations.
- Test with realistic data volumes and distributions.

  The test database must contain data representative of the production system. Tables must be fully populated and represent the data volume and cardinality between tables found in the production system. All production indexes must be built and the schema statistics must be populated correctly.
- Test with the optimizer mode to be used in production.

  Because Oracle Database research and development focuses on the query optimizer, Oracle re commends using the query optimizer in both the test and production environments.
- Test a single user performance first.

  Start testing with a single user on an idle or lightly-used database. If a single user cannot achieve acceptable performance under ideal conditions, then multiple users cannot achieve acceptable performance under real conditions.
- Get an execution plan for each SQL statement.

  Verify that the optimizer uses optimal execution plans, and that you understand the relative cost of each SQL statement in terms of CPU time and physical I/O. Identify heavily used transactions that would benefit most from tuning.
- Test with multiple users.

  This testing is difficult to perform accurately because user workload and profiles might not be fully quantified. However, you must test DML transactions in a multiuser environment to ensure that there are no locking conflicts or serialization problems.
- Test with a hardware configuration as close as possible to the production system.

Testing on a realistic system is particularly important for network latencies, I/O subsystem bandwidth, and processor type and speed. Testing on an unrealistic system can fail to reveal potential performance problems.

- Measure steady state performance.

  Each benchmark run must have a ramp-up phase, where users connect to the database and start using the application. This phase lets frequently cached data be initialized into the cache and lets single-execution operations (such as parsing) be completed before reaching the steady state condition. Likewise, each benchmark run must end with a ramp-down period, where resources are freed from the system and users exit the application and disconnect from the database.

See Also:

- *Oracle Database Performance Tuning Guide* for more information about performance tuning, workload testing, modeling, and implementation
- Automatic Database Diagnostic Monitor (ADDM)and SQL Advisors and Memory Advisors for more information about ADDM and SQL Tuning Advisor respectively
- *Oracle Database 2 Day + Performance Tuning Guide* for more information about tuning SQL statements using the SQL Tuning Advisor

3.5 Using Client Result Cache

Topics:

- About Client Result Cache
- Benefits of Client Result Cache
- Guidelines for Using Client Result Cache
- Setting Result Cache Integrity
- Client Result Cache Consistency
- Deployment-Time Settings for Client Result Cache
- Client Result Cache Statistics
- Validation of Client Result Cache
- Client Result Cache and Server Result Cache
- Client Result Cache Demo Files
- Client Result Cache Compatibility with Previous Releases

3.5.1 About Client Result Cache

Applications that use Oracle Database drivers and adapters built on OCI libraries—including C, C++, Java (JDBC-OCI), PHP, Python, Ruby, and Perl—can use the client result cache to improve response times of repetitive queries.

Client result cache enables client-side caching of SQL query (`SELECT` statement) result sets in client memory. Because retrieving results from a client process is faster than calling the database and rerunning the query, frequently run queries perform significantly faster when their results are cached. Client result cache also reduces the server CPU time that would have been used to process the query, thereby improving server scalability.

OCI statements from multiple sessions can match the same cached result set in the OCI process memory if they have similar schemas, SQL text, bind values, and session settings. If not, the query execution is directed to the server.

Client result cache is transparent to applications, and its cache of result set data is kept consistent with session or database changes that affect its result set data.

Applications that use the client result cache benefit from faster performance for queries that have cache hits. These applications use the cached result sets on clients or middle tiers.

Client result cache works with OCI features such as the OCI session pool, the OCI connection pool, DRCP, and OCI transparent application failover (TAF).

When using the client result cache, you must also enable OCI statement caching or cache statements at the application level.

Note:

Oracle Database 18.1 onwards, Client Result Cache supports dynamic binding

See Also:

- *Oracle Call Interface Programmer's Guide* for information about statement caching
- *Oracle Database JDBC Developer's Guide* for information about JDBC statement caching
- *Oracle Call Interface Programmer's Guide*
- *Oracle C++ Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide*
- *Oracle Database Performance Tuning Guide*
- *Oracle Database Concepts*
- SQL hints and `RESULT_CACHE` for clauses of `ALTER TABLE` and `CREATE TABLE`
- `RESULT_CACHE_MODE`
- Setting Result Cache Integrity
- `CLIENT_RESULT_CACHE_STAT$` view

3.5.2 Benefits of Client Result Cache

The benefits of the client result cache are:

- Client result cache is on the client, therefore a cache hit causes fetch (`OCIStmtFetch2()`) and execute (`OCIStmtExecute()`) calls to be processed locally, eliminating a server round trip. Eliminating server round trips reduces the use of server resources (such as server CPU and server I/O), significantly improving performance.

- Client result cache is transparent and consistent.
- Client result cache is available to every process, so multiple client sessions can simultaneously use matching cached result sets.
- Client result cache minimizes the need for each OCI application to have its own custom result cache.
- Client result cache transparently manages:
  - Concurrent access by multiple threads, multiple statements, and multiple sessions
  - Invalidation of cached result sets that might have been affected by database changes
  - Refreshing of cached result sets
  - Cache memory management
- Client result cache is automatically available to applications and drivers that use OCI libraries, including JDBC OCI, ODP.NET, OCCI, Pro*C/C++, Pro*COBOL, and ODBC.
- Client result cache uses OCI client memory, which might be less expensive than server memory.
- A local cache on the client has better locality of reference for queries executed by that client.

See Also:

`OCIStmtExecute()` and `OCIStmtFetch2()` in *Oracle Call Interface Programmer's Guide*

3.5.3 Guidelines for Using Client Result Cache

You can enable or disable the client result cache at three levels, which are, in order of descending precedence:

1. Query

   For a specific query, you can enable or disable the client result cache with a SQL hint. To add or change a SQL hint, you must change the application.
2. Table

   For all queries on a specific table, you can enable or disable the client result cache with a table annotation, without changing the application.
3. Session

   For all queries in your database session, you can enable or disable the client result cache with a session parameter, without changing the application.

Higher-level settings take precedence over lower-level ones.

Oracle recommends enabling the client result cache only for queries on read-only and seldom-updated tables (tables that are rarely updated). That is, enable the client result cache:

- At query level only for queries of read-only and seldom-read tables
- At table level only for read-only and seldom-read tables
- At session level only when running applications that query only read-only or seldom-read tables

Enabling the client result cache for queries that have large result sets or many sets of bind values can use a large amount of client result cache memory. Each set of bind values (for the same SQL text) creates a different cached result set.

When the client result cache is enabled for a query, its result sets can be cached on the client, server, or both. Client result cache can be enabled even if the server result cache (which is enabled by default) is disabled.

For OCI, the first `OCIStmtExecute()` call of every OCI statement handle call always goes to the server even if there might be a valid cached result set. An `OCIStmtExecute()` call must be made for each statement handle to be able to match a cached result set. Oracle recommends that applications either have their own statement caching for OCI statement handles or use OCI statement caching so that `OCIStmtPrepare2()` can return an OCI statement handle that has been executed once. Multiple OCI statement handles, from the same or different sessions, can simultaneously fetch data from the same cached result set.

For OCI, for a result set to be cached, the `OCIStmtExecute()` or `OCIStmtFetch2()` calls that transparently create this cached result set must fetch rows until ORA-01403 (No Data Found) is returned. Subsequent `OCIStmtExecute()` or `OCIStmtFetch2()` calls that match a locally cached result set need not fetch to completion.

See Also:

- SQL Hints
- Table Annotation
- Session Parameter
- Result Cache Mode Use Cases
- `OCIStmtExecute()`
- `OCIStmtPrepare2()`
- `OCIStmtFetch2()`

Topics:

- SQL Hints
- Table Annotation
- Session Parameter
- Effective Table Result Cache Mode
- Displaying Effective Table Result Cache Mode
- Result Cache Mode Use Cases
- Queries Never Result Cached in Client Result Cache

3.5.3.1 SQL Hints

The SQL hint `RESULT_CACHE` or `NO_RESULT_CACHE` applies to a single query, for which it enables or disables the client result cache. These hints take precedence over both table annotations and the `RESULT_CACHE_MODE` server initialization parameter.

For OCI, the SQL hint `/*+ result_cache */` or `/*+ no_result_cache */` must be set in SQL text passed to `OCIStmtPrepare()` and `OCIStmtPrepare2()` calls.

For JDBC OCI, the SQL hint `/*+ result_cache */` or `/*+ no_result_cache */` is included in the query (`SELECT` statement) as part of the query string.

See Also:

- *Oracle Database SQL Language Reference* for general information about SQL hints
- `OCIStmtPrepare()`, `OCIStmtPrepare2()` in *Oracle Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide* for information about SQL hints in JDBC

3.5.3.2 Table Annotation

A **table annotation** enables or disables the client result cache for all queries on a specific table. A table annotation takes precedence over the `RESULT_CACHE_MODE` server initialization parameter, but the SQL hints `/*+ RESULT_CACHE */` and `/*+ NO_RESULT_CACHE */` take precedence over a table annotation.

You annotate a table with either the `CREATE TABLE` or `ALTER TABLE` statement, using the `RESULT_CACHE` clause. To enable the client result cache, specify `RESULT_CACHE (MODE FORCE)`; to disable it, use `RESULT_CACHE (MODE DEFAULT)`.

Table 3-1 summarizes the table annotation result cache modes.

Table 3-1 Table Annotation Result Cache Modes

Mode	Description
`DEFAULT`	The default value. Result caching is not determined at the table level. You can use this value to clear any table annotations.
`FORCE`	If all table names in the query have this setting, then the query result is always considered for caching unless the `NO_RESULT_CACHE` hint is specified for the query. If one or more tables named in the query are set to `DEFAULT`, then the effective table annotation for that query is `DEFAULT`.

See Also:

- *Oracle Database SQL Language Reference* for more information about `RESULT_CACHE` clause of `ALTER TABLE` and `CREATE TABLE`
- *Oracle Database JDBC Developer's Guide* for information about table annotations in JDBC

3.5.3.3 Session Parameter

The session parameter RESULT_CACHE_MODE enables or disables the client result cache for all queries for your database session. RESULT_CACHE_MODE can be overruled for specific tables by table annotations and for specific queries by SQL hints.

You can set RESULT_CACHE_MODE in either the server parameter file (init.ora) or the ALTER SESSION or ALTER SYSTEM statement. To enable the client result cache, set RESULT_CACHE_MODE to FORCE; to disable it, set RESULT_CACHE_MODE to MANUAL.

See Also:

Oracle Database Reference for more information about RESULT_CACHE_MODE

3.5.3.4 Effective Table Result Cache Mode

The effective result cache mode for a table depends on both the table annotation result cache mode and the RESULT_CACHE_MODE session parameter setting, as Table 3-2 shows.

Table 3-2 Effective Result Cache Table Mode

Table Annotation Result Cache Mode	RESULT_CACHE_MODE = MANUAL	RESULT_CACHE_MODE = FORCE
FORCE	FORCE	FORCE
DEFAULT	MANUAL	FORCE

When the effective mode is FORCE, every query is considered for result caching unless the query has the NO_RESULT_CACHE hint, but actual result caching depends on internal restrictions for client and server caches, query potential for reuse, and space available in the client result cache.

The effective table result cache mode FORCE is similar to the SQL hint RESULT_CACHE in that both are only requests.

3.5.3.5 Displaying Effective Table Result Cache Mode

To display the result cache mode for one or more tables, see the RESULT_CACHE column of a *_TABLES static data dictionary view. For example:

- This query displays the result cache mode for the table T:

  ```
  SELECT result_cache FROM all_tables WHERE table_name = 'T'
  ```

- This query displays the result cache mode for all relational tables that you own:

  ```
  SELECT table_name, result_cache FROM user_tables
  ```

If the result cache mode is DEFAULT, then the table is not annotated.

If the result cache mode is FORCE, then the table was annotated with the RESULT_CACHE (MODE FORCE) clause of the CREATE TABLE or ALTER TABLE statement.

If the result cache mode is `MANUAL`, then the session parameter `RESULT_CACHE_MODE` was set to `MANUAL` in either the server parameter file (`init.ora`) or an `ALTER SESSION` or `ALTER SYSTEM` statement.

See Also:

Oracle Database Reference for more information about `*_TABLES` static data dictionary views

3.5.3.6 Result Cache Mode Use Cases

The following examples show when SQL hints take precedence over table annotations and the `RESULT_CACHE_MODE` session parameter.

- If the `emp` table is annotated with `ALTER TABLE emp RESULT_CACHE` (`MODE FORCE`) and the session parameter has its default value, `MANUAL`, then queries on `emp` are considered for result caching.

  However, results of the query `SELECT /*+ no_result_cache */ empno FROM emp` are not cached, because the SQL hint takes precedence over the table annotation and session parameter.
- If the `emp` table is not annotated, or is annotated with `ALTER TABLE emp RESULT_CACHE` (`MODE DEFAULT`), and the session parameter has its default value, `MANUAL`, then queries on `emp` are not result cached.

  However, results of the query `SELECT /*+ result_cache */ * FROM emp` are considered for caching, because the SQL hint takes precedence over the table annotation and session parameter.
- If the session parameter `RESULT_CACHE_MODE` is set to `FORCE`, and no table annotations or SQL hints override it, then all queries on all tables are considered for query caching.

3.5.3.7 Queries Never Result Cached in Client Result Cache

Results of the following queries are never cached in the client result cache (even if the queries include the `RESULT_CACHE` hint):

- Queries that contain any of the following:
  - Remote object
  - Complex type in the select list
  - PL/SQL function
- Snapshot-based queries
- Flashback queries
- Queries executed in serializable, read-only transactions
- Queries of tables on which virtual private database (VPD) policies are enabled

Such queries might be cached on the database if the server result cache feature is enabled—for more information, see *Oracle Database Concepts*.

3.5.4 Setting Result Cache Integrity

With Oracle Database 23ai, you can control the integrity of the result cache using the database initialization parameter: `RESULT_CACHE_INTEGRITY`. The `RESULT_CACHE_INTEGRITY` parameter enables you to specify whether the result cache must consider caching the queries that use potentially non-deterministic constructs, such as PL/SQL functions that are not declared as deterministic. In other words, you can use the `RESULT_CACHE_INTEGRITY` parameter to ensure that only explicitly deterministic queries are cached.

Enforcing the need to explicitly declare objects as deterministic before being considered for result caching can eventually improve the code quality while also rule out the chance of accidentally caching objects that should not be cached.

You can set the `RESULT_CACHE_INTEGRITY` parameter as `ENFORCED` or `TRUSTED`, which directs the database to do one of the following:

- When `RESULT_CACHE_INTEGRITY = ENFORCED`, enforce result cache integrity regardless of the setting of the `RESULT_CACHE_MODE` initialization parameter or any specified hints, allowing only deterministic constructs to be eligible for result caching. For example, queries using PL/SQL functions that are not explicitly declared as deterministic are not cached.
- When `RESULT_CACHE_INTEGRITY = TRUSTED`, honor the setting of the `RESULT_CACHE_MODE` initialization parameter and any specified hints, and consider queries that use possibly non-deterministic constructs as candidates for result caching. For example, queries using PL/SQL functions that are not explicitly declared as deterministic may be cached. Results that are certain to be non-deterministic (for example, `SYSDATE` or constructs involving `SYSDATE`) are not cached.

The `RESULT_CACHE_INTEGRITY` parameter defaults to `TRUSTED` for Oracle Database 23ai (unmanaged) and `ENFORCED` for Oracle Autonomous Database 23ai.

Note:

Setting the `RESULT_CACHE_INTEGRITY` parameter to `TRUSTED` achieves backward compatibility with databases prior to 23ai.

If the `RESULT_CACHE_MODE` parameter is set to `FORCE` or `FORCE_TEMP`, then Oracle recommends setting the `RESULT_CACHE_INTEGRITY` parameter to `ENFORCED`, which ensures that only deterministic constructs are eligible for result caching. This prevents the caching of non-deterministic constructs, which can potentially cause material changes to results.

See Also:

- `RESULT_CACHE_INTEGRITY` and `RESULT_CACHE_MODE` in *Database Reference*
- Setting the Result Cache Mode in *Database Performance Tuning Guide*
- SQL hints and `RESULT_CACHE` in *SQL Language Reference* for clauses of `ALTER TABLE` and `CREATE TABLE`

3.5.5 Client Result Cache Consistency

Client result cache transparently keeps its result sets consistent with any database or session state changes that affect them.

When a transaction modifies the data or metadata of any database object used to construct a cached result set, invalidation of that cached result set is sent to the client on its next round trip to the server. If the application does no database calls for a period of time, then the client result cache lag setting forces the next `OCIStmtExecute()` call to make a database call to check for such invalidations.

Cached result sets relevant to database invalidations are immediately invalidated. Any OCI statement handles that are fetching from cached result sets when their invalidations are received can continue fetching from them, but no subsequent `OCIStmtExecute()` calls can match them.

The next `OCIStmtExecute()` call by the process might cache the new result set if the client result cache has space available. Client result cache periodically reclaims unused memory.

If a session has a transaction open, OCI ensures that its queries that reference database objects changed in this transaction go to the server instead of the client result cache.

This consistency mechanism ensures that the client result cache is always close to committed database changes. If the application has relatively frequent calls involving database round trips due to queries that cannot be cached (DMLs, `OCILob` calls, and so on), then these calls transparently keep the client result cache consistent with database changes.

Sometimes, when a table is modified, a trigger causes another table to be modified. Client result cache is sensitive to such changes.

When the session state is altered—for example, when NLS session parameters are modified—query results can change. Client result cache is sensitive to such changes and for subsequent query executions, returns the correct result set. However, the client result cache keeps the current cached result sets (and does not invalidate them) so that other sessions in the process can match them. If other processes do not match them, these result sets are "Ruled" after a while. Result sets that correspond to the new session state are cached.

For an application, keeping track of database and session state changes that affect query result sets can be cumbersome and error-prone, but the client result cache transparently keeps its result sets consistent with such changes.

Client result cache does not require thread support in the client.

See Also:

`OCIStmtExecute()` in *Oracle Call Interface Programmer's Guide*

3.5.6 Deployment-Time Settings for Client Result Cache

When you deploy your application, you can set the following for the client result cache (without changing the application):

- Server initialization parameters
- Client configuration parameters

- Table annotations
- `RESULT_CACHE_MODE` session parameter

See Also:

- Server Initialization Parameters
- Client Configuration Parameters
- Table Annotation
- Session Parameter
- Client-Side Deployment Parameters

3.5.6.1 Server Initialization Parameters

The server initialization parameters to set for the client result cache when you deploy your application are:

- COMPATIBLE
- CLIENT_RESULT_CACHE_SIZE
- CLIENT_RESULT_CACHE_LAG

3.5.6.1.1 COMPATIBLE

Specifies the release with which Oracle Database must maintain compatibility. To enable the client result cache, `COMPATIBLE` must be at least 11.0.0.0. To enable client the result cache for views, `COMPATIBLE` must be at least 11.2.0.0.

3.5.6.1.2 CLIENT_RESULT_CACHE_SIZE

Specifies the maximum size of the client result set cache for each OCI client process. The default value, 0, means that the client result cache is disabled. To enable the client result cache, set `CLIENT_RESULT_CACHE_SIZE` to at least 32768 bytes (32 kilobytes (KB)).

If the client result cache is enabled on the server by `CLIENT_RESULT_CACHE_SIZE`, then its value can be overridden by the `sqlnet.ora` configuration parameter `OCI_RESULT_CACHE_MAX_SIZE`. If the client result cache is disabled on the server, then `OCI_RESULT_CACHE_MAX_SIZE` is ignored and the client result cache cannot be enabled.

Oracle recommends either enabling the client result cache for all Oracle Real Application Clusters (Oracle RAC) nodes or disabling the client result cache for all Oracle RAC nodes. Otherwise, within a client process, some sessions might have caching enabled and other sessions might have caching disabled (thereby getting the latest results from the server). This combination might present an inconsistent view of the database to the application.

`CLIENT_RESULT_CACHE_SIZE` is a static parameter. Therefore, if you use an `ALTER SYSTEM` statement to change the value of `CLIENT_RESULT_CACHE_SIZE`, you must include the `SCOPE = SPFILE` clause and restart the database before the change will take effect.

The maximum value for `CLIENT_RESULT_CACHE_SIZE` is the least of these values:

- Available client memory

- ((Possible number of result sets to be cached) * (average size of a row in a result set) * (average number of rows in a result set))
- 2 gigabytes (GB)

  If you specify a value greater than 2 GB, then the value is 2 GB.

> **Note:**
>
> Do not set the `CLIENT_RESULT_CACHE_SIZE` parameter during database creation, because that can cause errors.

3.5.6.1.3 CLIENT_RESULT_CACHE_LAG

Specifies the maximum time in milliseconds that the client result cache can lag behind changes in the database that affect its result sets. The default is 3000 milliseconds.

`CLIENT_RESULT_CACHE_LAG` is a static parameter. Therefore, if you use an `ALTER SYSTEM` statement to change the value of `CLIENT_RESULT_CACHE_LAG`, you must include the `SCOPE = SPFILE` clause and restart the database before the change will take effect.

3.5.6.2 Client Configuration Parameters

Client configuration parameters are optional, but if set, they override the equivalent parameters in the server initialization file `init.ora`.

Client configuration parameters can be set in the `oraaccess.xml` file, the `sqlnet.ora` file, or both. When equivalent parameters are set both files, the `oraaccess.xml` setting takes precedence over the corresponding `sqlnet.ora` setting. When a parameter is not set in `oraaccess.xml`, the process searches for its setting in `sqlnet.ora`.

When a client configuration parameter can be set in both `oraaccess.xml` and `sqlnet.ora`, Oracle recommends setting the parameter in `oraaccess.xml`. However, for a network configuration, `sqlnet.ora` is the primary file, because `oraaccess.xml` does not support network level settings.

Table 3-3 describes the equivalent `oraaccess.xml` and `sqlnet.ora` client configuration parameters.

Table 3-3 Client Configuration Parameters (Optional)

oraacess.xml Parameter	sqlnet.ora Parameter	Description
`max_size`	`OCI_RESULT_CACHE_MAX_SIZE`	Maximum size in bytes for the client result cache for each process. Specifying a size less than 32768 in `sqlnet.ora` disables client result cache for client processes the read `sqlnet.ora`.
`max_rset_size`	`OCI_RESULT_CACHE_MAX_RSET_SIZE`	Maximum size of any result set in bytes in the client result cache for each process.

Table 3-3 (Cont.) Client Configuration Parameters (Optional)

oraacess.xml Parameter	sqlnet.ora Parameter	Description
`max_rset_rows`	`OCI_RESULT_CACHE_MAX_RSET_ROWS`	Maximum size of any result set in rows in the client result cache for each process.

The result cache lag cannot be set on the client.

Related Topics

- *Oracle Call Interface Programmer's Guide*

3.5.7 Client Result Cache Statistics

On round trips to the server from the OCI client, OCI periodically sends the client result cache statistics to the server. These statistics, shown in the `CLIENT_RESULT_CACHE_STATS$` view, include number of result sets successfully cached, number of cache hits, and number of cached result sets invalidated. The number of cache misses for queries is at least equal to the number of Create Counts in the client result cache statistics. More precisely, the cache miss count equals the number of server executions in server Automatic Workload Repository (AWR) reports.

See Also:

- *Oracle Database Reference* for information about the `CLIENT_RESULT_CACHE_STAT$` view
- *Oracle Database Performance Tuning Guide* to find the client process IDs and session IDs for the sessions doing client result caching

3.5.8 Validation of Client Result Cache

Some ways to validate client result cache are:

- Measure Execution Times
- Query V$MYSTAT
- Query V$SQLAREA

3.5.8.1 Measure Execution Times

First, measure the execution time of the queries without `RESULT_CACHE` hints. Then add `RESULT_CACHE` hints to the queries and measure the execution time again. The difference in execution times is your performance gain.

3.5.8.2 Query V$MYSTAT

Note:

To query the `V$MYSTAT` view, you must have the `SELECT` privilege on it.

1. Run this query 5 times:

   ```
   SELECT count(*) FROM table_name
   ```

2. Query `V$MYSTAT`:

   ```
   SELECT * FROM V$MYSTAT
   ```

3. Run this query 5 times:

   ```
   SELECT /*+ result_cache */ count(*) FROM table_name
   ```

   Because the query results are cached, this step requires fewer round trips between client and server than `step 1` did.

4. Query `V$MYSTAT`:

   ```
   SELECT * FROM V$MYSTAT
   ```

   Compare the values of the columns for this query to those for the query in `step 2`.

Instead of adding the hint to the query in `step 3`, you can add the table annotation `RESULT_CACHE (MODE FORCE)` to *`table_name`* at `step 3` and then run the query in `step 1` a few times.

3.5.8.3 Query V$SQLAREA

Note:

To query the `V$SQLAREA` view, you must have the `SELECT` privilege on it.

1. Run this query 5 times:

   ```
   SELECT count(*) FROM table_name
   ```

2. Query `V$SQLAREA`:

   ```
   SELECT executions, fetches, parse_calls FROM V$SQLAREA
     WHERE sql_text LIKE '% FROM table_name'
   ```

3. Run this query 5 times:

   ```
   SELECT /*+ result_cache */ count(*) FROM table_name
   ```

4. Query `V$SQLAREA`:

   ```
   SELECT executions, fetches, parse_calls FROM V$SQLAREA
     WHERE sql_text LIKE '% FROM table_name'
   ```

Compare the values of the columns `executions`, `fetches`, and `parse_calls` for this query to those for the query in `step 2`. The difference in execution times is your performance gain.

Instead of adding the hint to the query in `step 3`, you can add the table annotation `RESULT_CACHE (MODE FORCE)` to *`table_name`* at step `step 3` and then run the query in step `step 1` a few times.

3.5.9 Client Result Cache and Server Result Cache

Client result cache is different from server result cache. Client result cache caches results of top-level SQL queries in OCI client memory, whereas the server result cache caches result sets and query fragments in server SGA memory.

You can enable the client result cache independently of the server result cache, though they share the result cache SQL hints, table annotations, and session parameter `RESULT_CACHE_MODE`. Table 3-4 shows the result cache association for result cache parameters, the PL/SQL package `DBMS_RESULT_CACHE`, and result cache views.

See Also:

Oracle Database Concepts

Table 3-4 Setting Client Result Cache and Server Result Cache

Parameters, PL/SQL Package, and Database Views	Result Cache Association
`CLIENT_RESULT_CACHE_*` parameters: • `CLIENT_RESULT_CACHE_SIZE` • `CLIENT_RESULT_CACHE_LAG`	client result cache
SQL hints: • `RESULT_CACHE` • `NO_RESULT_CACHE`	client result cache, server result cache
`sqlnet.ora OCI_RESULT_CACHE*` parameters: • `OCI_RESULT_CACHE_MAX_SIZE` • `OCI_RESULT_CACHE_MAX_RSET_SIZE` • `OCI_RESULT_CACHE_MAX_RSET_ROWS`	client result cache
`CLIENT_RESULT_CACHE_STATS$` view	client result cache
`RESULT_CACHE_MODE` parameter	client result cache, server result cache
All other `RESULT_CACHE_*` parameters (for example, `RESULT_CACHE_MAX_SIZE`)	server result cache
`DBMS_RESULT_CACHE` package	server result cache
`V$RESULT_CACHE_*` and `GV$RESULT_CACHE_*` views (for example, `V$RESULT_CACHE_STATISTICS` and `GV$RESULT_CACHE_MEMORY`)	server result cache
`CREATE TABLE` annotation	client result cache, server result cache
`ALTER TABLE` annotation	client result cache, server result cache

3.5.10 Client Result Cache Demo Files

For OCI applications, demonstration files for the client result cache are `cdemoqc.sql`, `cdemoqc.c`, and `cdemoqc2.c` (in the `demo` directory for your operating system).

3.5.11 Client Result Cache Compatibility with Previous Releases

To use the client result cache, applications must be relinked with Oracle Database 11*g* Release 1 (11.1) or later client libraries and be connected to an Oracle Database 11*g* Release 1 (11.1) or later database server. Client result cache is available to all OCI applications, including JDBC Type II driver, OCCI, Pro*C/C++, and ODP.NET. OCI drivers automatically pass the SQL hint `RESULT_CACHE` to `OCIStmtPrepare()` and `OCIStmtPrepare2()` calls.

See Also:

`OCIStmtPrepare()`, `OCIStmtPrepare2()` in *Oracle Call Interface Programmer's Guide*

3.6 Statement Caching

Statement caching is a feature that establishes and manages a cache of statements for each session. In the server, statement caching lets cursors be used without reparsing the statement, eliminating repetitive statement parsing. You can use statement caching with both connection pooling and session pooling, thereby improving performance and scalability. You can also use statement caching without session pooling in OCI and without connection pooling in OCCI, in the JDBC interface, and in the ODP.NET interface. You can also use dynamic SQL statement caching in Oracle precompiler applications that rely on dynamic SQL statements, such as Pro*C/C++ and ProCOBOL.

In the JDBC interface, you can enable and disable implicit and explicit statement caching independently of the other—you can use either, neither, or both. Implicit and explicit statement caching share a single cache for each connection. You can also disable implicit caching for a particular statement.

See Also:

- *Oracle Call Interface Programmer's Guide* for more information and guidelines about using statement caching in OCI
- *Oracle C++ Call Interface Programmer's Guide* for more information about statement caching in OCCI
- *Oracle Database JDBC Developer's Guide* for more information about using statement caching
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows* for more information about using statement caching in ODP.NET applications
- *Oracle Database Programmer's Guide to the Oracle Precompilers*, *Pro*C/C++ Programmer's Guide*, and *Pro*COBOL Programmer's Guide* for more information about using dynamic SQL statement caching in precompiler applications that rely on dynamic SQL statements

3.7 OCI Client Statement Cache Auto-Tuning

OCI client statement cache auto-tuning optimizes OCI client session features of middle-tier applications to improve performance without changing your OCI application.

Without auto-tuning, the OCI client statement cache size setting can become suboptimal—for example, when a changing workload causes a different working set of SQL statements. If the size is too low, it causes excess network activity and more parses at the server. If the size is too high, it causes excess memory use. It can be difficult for the client application to keep the cache size optimal.

Auto-tuning solves this potential performance problem by automatically and periodically reconfiguring the OCI statement cache size.

Auto-tuning is achieved by providing a deployment-time setting that provides an option to reconfigure OCI statement caching. These settings are provided as connect-string-based deployment settings in a client `oraaccess.xml` file that overrides programmatic settings to the user configuration of OCI features.

Middle-tier application developers and database administrators (DBAs) can expect reduced time and effort in diagnosing and fixing performance problems with each part of their system using the auto-tuning OCI client statement caching parameter setting.

See Also:

- *Oracle Call Interface Programmer's Guide*

3.8 Client-Side Deployment Parameters

Beginning with Oracle Database 12*c* Release 1 (12.1.0.1), OCI deployment parameters are available in a new configuration file (`oraaccess.xml`).

See Also:

Oracle Call Interface Programmer's Guide.

3.9 Using Query Change Notification

Continuous Query Notification (CQN) lets client applications register queries with the database and receive notifications of DML or DDL changes on the objects (object change notification (OCN)) or result set changes associated with the queries (query result change notification (QRCN)). The database publishes notifications when the DML or DDL transaction commits.

A **CQN registration** associates one or more queries with a notification type (OCN or QRCN) and a notification handler. To create a CQN registration, you can use:

- PL/SQL interface

  When you use the PL/SQL interface, the notification handler is a server-side PL/SQL stored procedure. PL/SQL registration can be used by nonthreaded languages and systems, such as PHP. For PHP, the PL/SQL listener invokes a PHP callback when it receives the database notification.
- Oracle Call Interface (OCI)

  When you use OCI, the notification handler is a client-side C callback procedure.
- Java Database Connectivity (JDBC) interface

  When you use the JDBC interface, the JDBC driver creates a registration on the server. The JDBC driver launches a new thread that listens for notifications from the server (through a dedicated channel), converts them to Java events, and then notifies all listeners registered with this registration.

See Also:

- Using Continuous Query Notification (CQN) for a complete discussion of the concepts of this feature and how to use the PL/SQL and OCI interfaces to create CQN registrations
- *Oracle Database JDBC Developer's Guide* for information about using JDBC for CQN registration on the server

3.10 Using Database Resident Connection Pool

Database Resident Connection Pool (DRCP) provides a connection pool in the database server for typical web application usage scenarios where the application acquires a database connection, works on the database for a relatively short time, and then releases the connection.

- About Database Resident Connection Pool
- Configuring DRCP
- Using Multi-pool DRCP
- Sharing Proxy Sessions

3.10.1 About Database Resident Connection Pool

DRCP offers a unique connection pooling solution that addresses scalability requirements in environments requiring large numbers of connections with minimal database resource usage.

DRCP pools server processes, each of which is the equivalent of a dedicated server process and database session combined; these are called **pooled servers**. Pooled servers can be shared by multiple applications running on the same or several hosts. A connection broker process manages the pooled servers at the database instance level.

DRCP is configurable at the PDB level, or can be chosen at application runtime. It allows concurrent use of traditional and DRCP-based connection architectures.

DRCP is especially useful for architectures with multiprocess single-threaded application servers (such as PHP and Apache) that cannot do middle-tier connection pooling. DRCP is also very useful in large-scale web deployments where hundreds or thousands of web servers or middle-tiers need database access and client-side pools (even in multithreaded systems and languages such as Java). Using DRCP, the database can scale to tens of thousands of simultaneous connections. If your database web application must scale to large numbers of connections, DRCP is your connection pooling solution.

DRCP complements middle-tier connection pools that share connections between threads in a middle-tier process. DRCP also enables sharing of database connections across middle-tier processes on the same middle-tier host, across multiple middle-tier hosts, and across multiple middle-tiers (web servers, containers) that accommodate applications written in different languages. This sharing significantly reduces the database resources needed to support a large number of client connections, thereby reducing the database tier memory footprint and increasing the scalability of both middle and database tiers. Having a pool of readily available servers also reduces the cost of creating and releasing client connections.

Clients get connections from the DRCP, which is connected to an Oracle Database background process called the connection broker. The connection broker implements the pool functionality and multiplexes pooled servers among persistent inbound connections from the client.

When a client needs database access, the connection broker gets a server process from the pool and gives it to the client. The client is then directly connected to the server. After the server executes the client request, the server process returns to the pool and the connection from the client is restored to the connection broker as a persistent inbound connection from the client process. In DRCP, releasing resources leaves the session intact, but no longer associated with a connection (server process). Because this session stores its user global area (UGA) in the program global area (PGA), not in the system global area (SGA), a client can reestablish a connection transparently upon detecting activity.

DRCP is typically recommended for applications with a large number of connections. Shared servers are recommended for applications with a medium number of connections and dedicated sessions are recommended for applications with a small number of connections. The threshold sizes depend on the amount of memory available on the database host.

DRCP has these advantages:

- DRCP enables resource sharing among multiple client applications and middle-tier application servers.
- DRCP improves scalability of databases and applications by reducing resource usage on the database host.

Compared to client-side connection pooling and shared servers:

- DRCP provides a direct connection to the database server, furnished by client-side connection pooling (like client-side connection pooling but unlike shared servers).
- DRCP can pool database servers (like client-side connection pooling and shared servers).
- DRCP can pool sessions (like client-side connection pooling but unlike shared servers).
- DRCP can share connections across middle-tier boundaries (unlike client-side connection pooling).

See Also:

- *Oracle Database Concepts* for details about DRCP architecture
- *Oracle Database Administrator’s Guide* for more information about switch service enhancements.

3.10.2 Configuring DRCP

You can configure the DRCP at either the CDB or PDB level.

The database parameter `ENABLE_PER_PDB_DRCP` controls whether DRCP is configured in CDB DRCP mode or per-PDB DRCP mode.

You can configure Database Resident Connection Pool (DRCP) to use a CDB wide pool by setting the database parameter `ENABLE_PER_PDB_DRCP=FALSE`. The database administrator must start and configure DRCP using the `DBMS_CONNECTION_POOL` package in the ROOT container. In CDB DRCP mode, you cannot use subprograms in the `DBMS_CONNECTION_POOL` package to manage the pool at the PDB level. You can monitor pools usage and statistics in the ROOT container pertaining to the CDB by querying the `GV$CPOOL_STATS`, `GV$CPOOL_CC_STATS`, `GV$SPOOL_CONN_INFO`, `GV$AUTHPOOL_STATS` and `GV$SPOOL_CC_INFO` views. The SYS user in the PDB can view statistics from `GV$SPOOL_CC_INFO` and `GV$AUTHPOOL_STATS`. Configuration options include minimum and maximum number of pooled servers, number of connection brokers, maximum number of connections that each connection broker can handle, and so on.

Alternatively, the database administrator can enable the per-PDB DRCP mode by setting the database parameter `ENABLE_PER_PDB_DRCP=TRUE`. In this mode, you can create an isolated set of pooled servers for each pluggable database. The DRCP connection broker is started by the ROOT container. The PDB administrator can configure, manage and monitor independent pools customized to each PDB specific needs using the `DBMS_CONNECTION_POOL` subprograms. Processes are not shared between PDBs. ROOT and PDBs only share metadata. They do not share data in pool configuration tables. Each container stores its own pool configuration data. The statistics in the views will be kept for each PDB. The PDB administrator can neither alter the broker parameters, such as `MINSIZE`, `MAXCONN_CBROK` and `NUM_CBROK`, nor can they set the broker parameters to 2147483647. These parameters can only be modified in the database parameter `CONNECTION_BROKERS` and apply to the entire CDB. The database parameters `MIN_AUTH_SERVERS`, `MAX_AUTH_SERVERS` and `DRCP_DEDICATED_OPT` are dynamically per PDB modifiable using the `ALTER SYSTEM` command.

To avoid configuration, by default, DRCP auto-starts when a PDB opens, and it stops when the PDB closes. If the PDB administrator stops the pool, then the DRCP does not auto-start when the PDB re-opens in the future.

The `DRCP_DEDICATED_OPT` parameter controls the use of dedicated optimization with DRCP. It is disabled by default when per-PDB mode is enabled.

The `MAX_TXN_THINK_TIME` parameter specifies the maximum time of inactivity allowed before termination for a client with an open transaction. This can be set to allow more time than the client that does not have transactions (specified by the `MAX_THINK_TIME` parameter value). This allows efficient pool reuse, while giving incomplete transactions a longer time to conclude.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for details about the `DBMS_CONNECTION_POOL` package
- *Oracle Database Administrator's Guide* for more information about Configuring the Connection Pool for Database Resident Connection Pooling
- *Oracle Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer's Guide*
- *Oracle Universal Connection Pool for JDBC Developer's Guide*

3.10.2.1 Managing Permissions for Per-PDB DRCP

For the per-PDB DRCP configuration (`ENABLE_PER_PDB_DRCP=TRUE`), the `SYS` user must grant permissions to the `PDB` administrator to manage DRCP at the PDB level.

By default, DRCP is configured at the CDB level. In the CDB DRCP mode, a single DRCP pool running in the CDB is shared across all the PDBs. In the CDB DRCP mode, the `ENABLE_PER_PDB_DRCP` database initialization parameter is set to `FALSE`. The `ENABLE_PER_PDB_DRCP` parameter can be set to `TRUE` to enable per-PDB DRCP.

For a PDB administrator to access the `DBMS_CONNECTION_POOL` package and query the DRCP statistics, the ROOT user (SYS) must grant the following permissions to the PDB administrator (`PDB1ADMIN` in this case).

```
GRANT CREATE SESSION, CREATE SYNONYM TO PDB1ADMIN;
GRANT EXECUTE ON DBMS_CONNECTION_POOL TO PDB1ADMIN;
GRANT SELECT ON V_$CPOOL_STATS TO PDB1ADMIN;
GRANT SELECT ON V_$CPOOL_CC_STATS TO PDB1ADMIN;
GRANT SELECT ON V_$CPOOL_CONN_INFO TO PDB1ADMIN;
GRANT SELECT ON V_$CPOOL_CC_INFO TO PDB1ADMIN;
GRANT SELECT ON V_$AUTHPOOL_STATS TO PDB1ADMIN;
```

To make the management and monitoring of DRCP pools easier, the PDB administrator (`PDB1ADMIN` in this case) can create the following synonyms.

```
CREATE SYNONYM DBMS_CONNECTION_POOL FOR SYS.DBMS_CONNECTION_POOL;
CREATE SYNONYM V$CPOOL_STATS FOR SYS.V_$CPOOL_STATS;
CREATE SYNONYM V$CPOOL_CC_STATS FOR SYS.V_$CPOOL_CC_STATS;
CREATE SYNONYM V$CPOOL_CONN_INFO FOR SYS.V_$CPOOL_CONN_INFO;
CREATE SYNONYM V$CPOOL_CC_INFO FOR SYS.V_$CPOOL_CC_INFO;
CREATE SYNONYM V$AUTHPOOL_STATS FOR SYS.V_$AUTHPOOL_STATS;
```

Once this is done, pool management is allowed only at the PDB level by the respective PDB administrators.

3.10.3 Using Multi-pool DRCP

Starting Oracle Database 23ai, you can use multiple, named DRCP pools. Database administrators can add, configure, manage, monitor, or remove a DRCP pool at the PDB or CDB level. You can configure DRCP to use connections (pooled servers) from any available DRCP pool and have a specific application acquire connections from a configured DRCP pool.

The default system-named pool: `SYS_DEFAULT_CONNECTION_POOL` is always available. You can create a new, named pool using the `ADD_POOL` procedure in the `DBMS_CONNECTION_POOL` package. Depending on your requirements, applications can use connections from any DRCP pool.

Having multiple pools allows finer control over the DRCP pool usage. You can have pooled servers available to a few applications or services at all times. You can avoid a situation where connections from some applications occupy all the pooled servers of a DRCP pool while other applications wait for an available pooled server in that pool.

See Also:

- Adding a DRCP Pool
- Removing a DRCP Pool
- Connecting to a Pool in Multi-pool DRCP

Components of Multi-pool DRCP

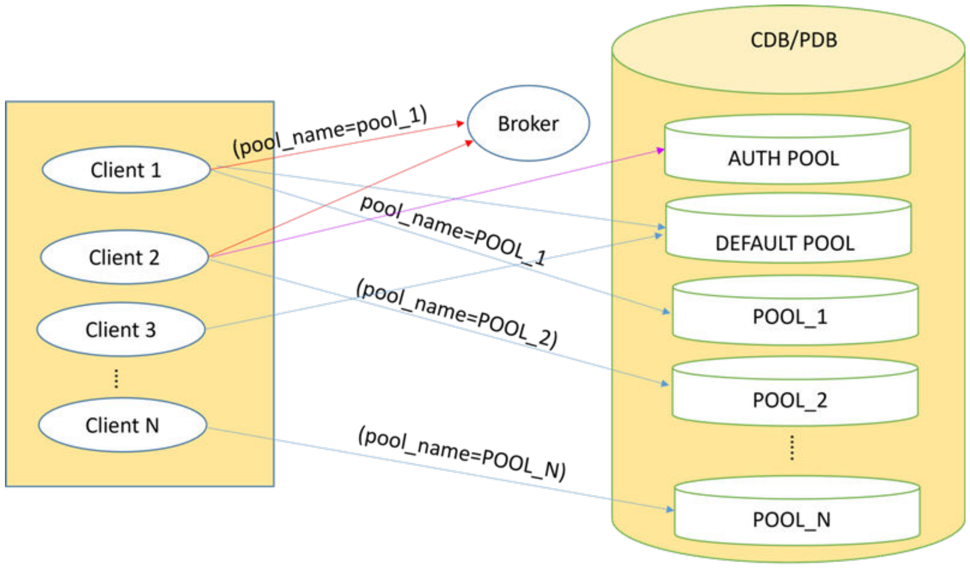
The following components are common to all the multi-pool DRCP pools and the default pool, at the PDB or CDB level.

- A connection broker to manage the pooled servers and handle the connection hand-off process.
- An authentication pool to authenticate user connections when client applications connect to DRCP.

Other components include:

- A default pool called `SYS_DEFAULT_CONNECTION_POOL` to handle DRCP for pools when no pool name is specified. The client cannot add or remove the default pool.
- Multiple DRCP pools that are added at the PDB or CDB level.

Figure 3-1 Multi-pool DRCP



3.10.3.1 Adding a DRCP Pool

By connecting to a PDB, a PDB administrator can add a DRCP pool at the PDB level. Similarly, a CDB administrator can add a DRCP pool at the CDB level.

To add a new pool, use the `dbms_connections_pool.add_pool()` procedure.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for details about the `DBMS_CONNECTION_POOL` package

You must add a unique name for the pool. For example, you cannot add a new pool called `SYS_DEFAULT_CONNECTION_POOL` because it is the default pool name.

Multi-pool DRCP has the same default configuration values as the per-PDB DRCP, such as `minsize=0`, `num_cbrok=0`, and `maxconn_cbrok=0`. If the configuration is known at the time of adding the pool, you can use the `dbms_connections_pool.add_pool()` procedure to set the configuration while adding the pool itself. If the configuration is not known, then it must be reconfigured later using the `configure_pool()` procedure with the new pool name and its configuration values.

You can use the `start_pool()`, `stop_pool()`, `configure_pool()`, `alter_param()` and `restore_defaults()` procedures of the `dbms_connection_pool` package for the newly added pools.

The `V$CPOOL_STATS`, `V$CPOOL_CC_STATS`, `V$CPOOL_CONN_INFO`, and `V$CPOOL_CC_INFO` views have the `POOL_NAME` columns and provide the statistics on the added pools.

Example 3-1   Adding a DRCP pool

```
exec dbms_connection_pool.add_pool('mypool')
```

3.10.3.2 Removing a DRCP Pool

You must be a PDB or CDB administrator to remove a DRCP pool.

To remove a DRCP pool, use the `dbms_connections_pool.remove_pool()` procedure.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for details about the `DBMS_CONNECTION_POOL` package

You cannot use the `remove_pool` procedure to remove:

- The default pool.
- A pool that is currently active. You must stop the pool before removing it.
- A pool that does not exist.

Example 3-2   Removing a DRCP pool

```
exec dbms_connection_pool.remove_pool('mypool')
```

3.10.3.3 About Authentication Pool in Multi-pool DRCP

An authentication pool is common to all the added pools in multi-pool DRCP and the default pool (`SYS_DEFAULT_CONNECTION_POOL`) for CDB-wide and per-PDB pools. When the first pool in a multi-pool DRCP starts functioning, the authentication pool also starts functioning. The authentication pool stops functioning when the last `ACTIVE` pool stops functioning.

The statistics on `num_authentications` (number of authentications) for each pool is maintained in the `v$cpool_stats` and `v$cpool_cc_stats` views. The statistics about the authentication pool is provided in the `V$AUTHPOOL_STATS` view.

3.10.3.4 Managing the Connection Broker in Multi-pool DRCP

The ROOT container owns the connection broker, hence, with per-PDB DRCP, a broker process is always running. For CDB-wide pools, the database administrator can configure brokers using the `DBMS_CONNECTION_POOL` package but because the broker is at the ROOT-level and common to all the pools, changing the number of brokers for one pool affects all the other pools, which is fine as DBA is the same for all pools, unlike per-PDB pools. With per-PDB pools, a PDB administrator cannot alter or configure brokers using the `DBMS_CONNECTION_POOL` package because it affects other per-PDB pools. Therefore, it is recommended that for per-PDB pools, a CDB administrator must set brokers using the `CONNECTION_BROKERS` in the database initialization parameter.

Example:

```
alter system set CONNECTION_BROKERS='((TYPE=POOLED)(BROKERS=1)(CONNECTIONS=40000))';
```

3.10.4 Sharing Proxy Sessions

Starting with Oracle Database 12c Release 2 (12.2.0.1), proxy sessions in DRCP are shared among applications that are connected to the same schema.

3.10.5 Using JDBC with DRCP

Oracle JDBC drivers support DRCP.

The DRCP implementation creates a pool on the server side, which is shared across multiple client pools. These client pools use Universal Connection Pool for JDBC. Using Universal Connection Pool significantly lowers memory consumption (because of the reduced number of server processes on the server) and increases the scalability of the Database server.

To track check-in and checkout operations of server-side connections, Java applications must use a client-side pool such as Universal Connection Pool for JDBC or a third-party Java connection pool.

To enable DRCP on the client side, you must do the following:

- Pass a non-`NULL`, nonempty `String` value to the connection property `oracle.jdbc.DRCPConnectionClass`.
- Pass (`SERVER=POOLED`) in the long connection string.

You can also specify (`SERVER=POOLED`) in the short URL form as follows:

```
jdbc:oracle:thin:@//<host>:<port>/<service_name>[:POOLED]
```

For example:

```
jdbc:oracle:thin:@//localhost:5221/orcl:POOLED
```

By setting the same DRCP Connection class name for all the pooled server processes on the server using the connection property `oracle.jdbc.DRCPConnectionClass`, you can share pooled server processes on the server across multiple connection pools.

In DRCP, you can also apply a tag to a given connection and easily retrieve that tagged connection later.

See Also:

- Starting Database Resident Connection Pool
- *Oracle Database JDBC Developer's Guide* for more information about APIs that are used to control custom connection pool implementations
- *Oracle Database JDBC Developer's Guide* for more information about enabling DRCP on client side

3.10.6 Using OCI Session Pool APIs with DRCP

The OCI session pool APIs `OCISessionPoolCreate()`, `OCISessionGet()`, and `OCISessionRelease()` interoperate with DRCP.

An OCI application initializes the environment for the OCI session pool for DRCP by invoking `OCISessionPoolCreate()`, which is described in *Oracle Call Interface Programmer's Guide*.

To get a session from the OCI session pool for DRCP, an OCI application invokes `OCISessionGet()`, specifying `OCI_SESSGET_SPOOL` for the `mode` parameter.

To release a session to the OCI session pool for DRCP, an OCI application invokes `OCISessionRelease()`.

To improve performance, the OCI session pool can transparently cache connections to the connection broker. An OCI application can reuse the sessions within which the application leaves sessions of a similar state either by invoking `OCISessionGet()` with the `authInfop` parameter set to `OCI_ATTR_CONNECTION_CLASS` and specifying a connection class name or by using the `OCIAuthInfo` handle before invoking `OCISessionGet()`.

DRCP also supports features offered by the traditional client-side OCI session pool, such as tagging, statement caching, and TAF.

See Also:

- *Oracle Call Interface Programmer's Guide* for information about `OCISessionGet()`
- *Oracle Call Interface Programmer's Guide* for more information about `OCISessionRelease()`

3.10.7 Session Purity

Session purity specifies whether an OCI application can reuse a pooled session (`OCI_SESSGET_PURITY_SELF`) or must use a new session (`OCI_SESSGET_PURITY_NEW`).

The application can set session purity either on the `OCIAuthInfo` handle before invoking `OCISessionGet()` or in the `mode` parameter when invoking `OCISessionGet()`.

Example 3-3 Setting Session Purity for New Session

This example shows how a connection pooling application sets up a new session.

```
/* OCIAttrSet method */
ub4 purity = OCI_ATTR_PURITY_NEW;
OCIAttrSet(authInfop, OCI_HTYPE_AUTHINFO, &purity, (ub4)sizeof(purity) OCI_ATTR_PURITY,
errhp);

/*  OCISessionGet mode method */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0, NULL,
NULL, NULL, OCI_SESSGET_SPOOL);
/* poolName is the name returned by OCISessionPoolCreate() */
```

Note:

When reusing a pooled session, the NLS attributes of the server override those of the client.

For example, if the client sets `NLS_LANG` to `french_france.us7ascii` and then is assigned a German session from the pool, the client session becomes German.

To avoid this problem, use connection classes to restrict sharing.

3.10.8 Connection Class

Connection class defines a logical name for the type of connection that an OCI application needs. When a pooled session has a connection class, OCI ensures that the session is not shared outside of that connection class.

For example, a connection class can prevent the following from sharing pooled sessions:

- Different users

  (A session first created for user `HR` is assigned only to subsequent requests by user `HR`.)
- Different sessions of the same user
- Different applications being run by the same user

  (Each application can have its own connection class.)

To set the connection class, you can use the `OCI_ATTR_CONNECTION_CLASS` attribute of the `OCIAuthInfo` handle. A connection class name is a string of at most 1024 bytes, and it cannot include an asterisk (*).

3.10.8.1 Example: Setting the Connection Class as HRMS

You can use the `OCISessionPoolCreate` API to set a connection class as HRMS.

Example 3-4 specifies that an HRMS application needs sessions with the connection class `HRMS`.

Example 3-4 Setting the Connection Class as HRMS

```
OCISessionPoolCreate (envhp, errhp, spoolhp, &poolName, &poolNameLen, "HRDB",
    strlen("HRDB"), 0, 10, 1, "HR", strlen("HR"), "HR", strlen("HR"),
    OCI_SPC_HOMOGENEOUS);

OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO, "HRMS", strlen ("HRMS"),
    OCI_ATTR_CONNECTION_CLASS, errhp);
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
    NULL, NULL, NULL, OCI_SESSGET_SPOOL);
```

3.10.9 Session Purity and Connection Class Defaults

Table 3-5 shows the defaults for the attributes and settings of connections that an OCI application gets from the OCI session pool (using `OCISessionGet()`) and from other sources.

Table 3-5 Session Purity and Connection Class Defaults

Attribute or Setting	Default Value for Connection From OCI Session Pool	Default Value for Connection Not From OCI Session Pool
`OCI_ATTR_PURITY`	`OCI_ATTR_PURITY_SELF`	`OCI_ATTR_PURITY_NEW`
`OCI_ATTR_CONNECTION_CLASS`	OCI-generated globally unique name for each client-side session pool, used as the default connection class for all connections in the OCI session pool	`SHARED`
Sessions shared by ...	Threads that request sessions from the OCI session pool	Connections to a particular database that have the `SHARED` connection class

3.10.10 Setting the Purity and Connection Class in the Connection String

You can specify the connection class and purity attributes in the `CONNECT_DATA` section of the Connect strings.

When it is not possible to change the application setting in the application code, you can override the value set in the application by setting the parameters: `POOL_CONNECTION_CLASS` and `POOL_PURITY` in the Connect string.

Note:

If the `POOL_PURITY` is specified as `SELF` in the Connect string, applications that explicitly get `NEW` purity connections from the OCI Session Pool do not drop the DRCP pooled session while releasing the connection back to the OCI Session Pool, even if they specify the `OCI_SESSRLS_DROPSESS` mode. Such applications should continue to use the programmatic way of specifying the purity.

See Also:

- *Oracle Database Net Services Reference* for more information about DRCP parameters

3.10.11 Starting DRCP

The DBA must log on as `SYSDBA` and start the default pool, `SYS_DEFAULT_CONNECTION_POOL`, using `DBMS_CONNECTION_POOL.START_POOL` with the default settings.

See Also:

- *Oracle Database Administrator's Guide* for detailed information about configuring the pool

3.10.12 Shut Down Connection Draining for DRCP

You can shut down (close) a DRCP pool using the `DBMS_CONNECTION_POOL.STOP_POOL()` procedure in either of the following ways.

- If a DRCP pool is idle and inactive after your application has released all the connections back to the DRCP pool, use the `STOP_POOL` procedure using the pool name as the parameter to close the idle or inactive pool.
- If a DRCP pool has active connections, use an optional `draintime` parameter in the `STOP_POOL` procedure to close the connections and DRCP pool without waiting for the DRCP pool to go idle and inactive.

You can use the `draintime` parameter in the `STOP_POOL` procedure to close an active DRCP pool immediately, or after a specified connection drain time. The `draintime` parameter indicates how many seconds a DRCP pool is allowed to remain active before the pool is drained of its connections and the pool is closed.

For example, to allow 20 seconds for active connections in a DRCP pool to complete their tasks before being closed, run the following:

```
DBMS_CONNECTION_POOL.STOP_POOL('my_pool', 20);
```

To close a DRCP pool immediately, set the `draintime` parameter to 0, as follows:

```
DBMS_CONNECTION_POOL.STOP_POOL('my_pool', 0);
```

The ability to close active connection pools provides the DBAs better control over the DRCP usage and configurations.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for details about the `DBMS_CONNECTION_POOL` package

3.10.13 Enabling DRCP

To enable DRCP in an application, specify either `:POOLED` in the Easy Connect string (as in Example 3-5) or `(SERVER=POOLED)` in the TNS Connect string (as in Example 3-6).

Example 3-5 Enabling DRCP With `:POOLED` in the Easy Connect string

```
oraclehost.company.com:1521/books.company.com:POOLED
```

Example 3-6 Enabling DRCP With `SERVER=POOLED` in the TNS Connect string

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oraclehost.company.com)
 (PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)(SERVER=POOLED)))
```

3.10.14 Connecting to a Pool in Multi-pool DRCP

You can specify a pool name for each connection.

To access multi-pool DRCP and to connect with an appropriate pool, specify `POOL_NAME=<pool_name>` along with `SERVER=POOLED` in the connect string:

```
(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=<host>)(PORT=<port>))
         (CONNECT_DATA=(SERVER=POOLED)(POOL_NAME=<pool_name>)))
```

Note:

The connect string must include `SERVER=POOLED` for `POOL_NAME=<pool_name>` to work.

An application can connect to any of the available pools. DRCP checks if the pool name (`<pool_name>`) in the connect string exists, and if it does, the connection uses the pooled server with the given pool name. If the pool name does not exist, an error is returned. If no pool name is specified in the connect string, the connection is handed off to the default pool, provided it is active.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_CONNECTION_POOL` package
- Using Multi-pool DRCP

Example 3-7 Connecting to a Pool in Multi-pool DRCP

```
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=phoenix92128)(PORT=1521))
         (CONNECT_DATA=(SERVICE_NAME=cdb1_pdb1.regress.rdbms.dev.us.oracle.com)
         (SERVER=POOLED)(POOL_NAME=mypool)))
```

3.10.15 Implicit Connection Pooling

Implicit connection pooling is aimed at enhancing the benefits derived from DRCP and Proxy Resident Connection Pooling (PRCP). With implicit connection pooling, a database session from DRCP/PRCP gets automatically mapped to and unmapped from an application connection at runtime. For example, for a transaction, an application implicitly uses or reuses an available connection from the connection pool and releases the connection back to the pool after the transaction is complete.

See Also:

- *Implicit Connection Pooling with CMAN-TDM and PRCP* below for more information about PRCP.

You can use implicit connection pooling without making any application-level change or calling the pooling APIs. The configuration for implicit connection pooling is performed on the client side. You must provide the `POOL_BOUNDARY` parameter in the `CONNECT_DATA` section of the connect string. After configuring the `POOL_BOUNDARY` parameter on DRCP/PRCP, session mapping or unmapping is performed based on the session state. Implicit connection pooling uses a session's state and specific application settings to automatically detect the time (boundary) to unmap and release a connection back to the pool.

See Also:

- *Implicit Stateful and Stateless Sessions* below for more information about session states.

Implicit connection pooling provides better scalability and enables efficient use of database resources for applications that do not use application connection pools, such as Oracle Call Interface (OCI) Session Pool or Java Database Connectivity (JDBC) Oracle Universal Connection Pool (UCP).

Implicit connection pooling is beneficial to on-premise and cloud applications in the following cases:

- Applications that directly connect to Oracle Database and have pooling requirements but do not leverage the server-side connection pooling APIs of Oracle.
- Applications that are connected to Connection Manager-Traffic Connector Mode (CMAN-TDM) in PRCP and have pooling requirements but do not use the connection pooling APIs. These applications can configure TDM in PRCP mode to use Implicit Connection Pooling.
- Applications that use connection pooling APIs but need to further optimize the use of shared resources. Implementing implicit connection pools provides better scalability because applications do not hold up sessions unnecessarily while waiting for an explicit API call.
- Middle-tier servers can use Implicit Connection Pooling for better scalability through implicit multiplexing of database sessions across a larger number of middle-tier connections.

Implicit connection pooling is available to clients on Oracle Database 23ai and to applications that use any data access drivers, such as OCI, JDBC, ODP.Net, cx_Oracle (Python), node-oracledb (Node.js), PHP-OCI8, Pre-compilers, ODBC, and OCCI.

Topics

- Implicit Stateful and Stateless Sessions

3.10.15.1 Implicit Stateful and Stateless Sessions

A session is implicitly stateless if an active session's state satisfies all the following conditions:

- All the cursors that are open in the session have been fetched through to completion.
- The session has no active transactions.
- The session has no temporary LOBs.
- The session has no global temporary tables with rows.
- The session does not have private temporary tables open.

If a session state does not satisfy any of the aforesaid conditions, the session is implicitly stateful.

3.10.15.2 Statement and Transaction Boundary

Implicit Connection Pooling uses time boundaries to release a session back to the connection pool. A time boundary is a point in time in the life cycle of the application when an application session is released back to the pool. The pool could either be a DRCP pool or PRCP pool.

The two boundaries used in Implicit Connection Pooling are Statement Boundary and Transaction Boundary.

- Statement Boundary is used to release a session back to the connection pool when the session is implicitly stateless.

  A session is implicitly stateless when all open cursors in a session have been fetched through to completion, and there are no active transactions, temporary tables, or temporary LOBs.
- Transaction Boundary is used to release a session back to the connection pool when a transaction ends implicitly or explicitly, or when a transaction is not available and the session is stateless.

Note:

- The release to the connection pool closes any active cursors, temporary tables, and temporary LOBs.
- After a statement is executed or a temporary LOB is created and a commit or rollback is performed within the transaction boundary:
  - Subsequent fetch operations may encounter an invalid cursor-related error such as ORA-01001.
  - Subsequent use of temporary LOB may encounter an error such as ORA-22922.

  Working with a persistent LOB can continue even after an implicit release. However, if an implicit operation lands on a different instance, an ORA-43887 error may occur. In such cases, the operation should be re-attempted.

3.10.15.3 Configuring Implicit Connection Pool Boundaries

In the application tier, to enable DRCP pooling, the client must specify the server type as `POOLED` (`SERVER=POOLED`) in the connection string of the `tnsnames.ora` file.
To configure the Implicit Connection Pooling boundary for DRCP on the server side or for PRCP in the TDM mode, add the `POOL_BOUNDARY` attribute in the connection string. The attribute takes two valid values, namely `STATEMENT` and `TRANSACTON`, and has no default value. If the `POOL_BOUNDARY` attribute is not included in the connection string, Implicit Connection Pooling is disabled.

Note:

If the application provides the `POOL_BOUNDARY=STATEMENT` or `POOL_BOUNDARY=TRANSACTION` attribute in the connection string without providing the `SERVER=POOLED` attribute, Implicit Connection Pooling is disabled and the `POOL_BOUNDARY` directive is ignored.

Tip:

Use the Net Configuration Assistant (netca) utility to add the `POOL_BOUNDARY` attribute into the connection string.

- To specify the Statement Boundary, use `POOL_BOUNDARY=STATEMENT` in the Easy Connect string or in the TNS Connect string, as shown in the following examples:

Note:

The Easy Connect string syntax is supported for Oracle Database 23ai, and later releases.

```
inst1=(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=slc11xgx)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=t1.regress.rdbms.dev.us.oracle.com)(SERVER=POOLED)
(POOL_BOUNDARY=STATEMENT)))
```

```
host:port/servicename:pooled?pool_boundary=statement
```

- To specify the Transaction Boundary, use `POOL_BOUNDARY=TRANSACTIION` in the Easy Connect string or in the TNS Connect string, as shown in the following examples:

```
inst1=(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=slc11xgx)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=t1.regress.rdbms.dev.us.oracle.com)(SERVER=POOLED)
(POOL_BOUNDARY=TRANSACTION)))
```

```
host:port/servicename:pooled?pool_boundary=transaction
```

3.10.15.4 Impact of Round-trip OCI Calls on Implicit Connection Pooling States

If an application uses the session pooling APIs along with the `POOL_BOUNDARY=STATEMENT` or `TRANSACTION` attribute in the connection string, then the connection string settings take precedence over the pooling APIs. The session is released back to the DRCP or PRCP pool using the statement or transaction boundary directive, overriding the session release API call. For instance, if an `OCISessionRelease()` call is made on a session that was implicitly released using the `POOL_BOUNDARY=STATEMENT` or `TRANSACTION` attribute in the connection string, then the `OCISessionRelease()` call is a no-op. Conversely, if an `OCISessionRelease()` call is made when the session is implicitly stateful, and hence, not implicitly released, then the session is released back to the pool based on the API call.

Note:

For all the `POOL_BOUNDARY` options, the default purity is set to `SELF`. You can specify the purity using the `POOL_PURITY` parameter in the connect string to override the default purity value.

See Also:

Oracle Call Interface Programmer's Guide for more information about pooling options and OCI round-trip calls

3.10.15.5 Deciding which Pool Boundary to Use

Use `POOL_BOUNDARY` with `TRANSACTION` for applications that create many session states from partially fetched cursors, temporary LOBs, and global or private temporary tables and for applications that have occasional commits or rollbacks of transactions.

Use `POOL_BOUNDARY` with `STATEMENT` for applications that create minimal session states from partially fetched cursors, temporary LOBs, and global or private temporary tables.

3.10.15.6 Implicit Connection Pooling with CMAN-TDM and PRCP

Implicit Connection Pooling enables applications to leverage PRCP if Oracle Connection Manager (CMAN) is set to Traffic Director Mode (TDM). For applications that connect through the CMAN-TDM connection proxy, Implicit Connection Pooling with PRCP can help maximize the pooled server usage and reduce the server resource usage.

CMAN is a connection concentrator from Oracle Net Services that enables applications to connect to Oracle Database over the Cloud. TDM is an added layer on top of CMAN that provides a shield to the applications from database instance outages and takes care of failed connections due to outages.

You can configure CMAN to operate in the TDM mode. CMAN-TDM is an Oracle Database connection proxy that enables a client application to connect to an Oracle Database (on-premises and cloud) without exposing the underlying database details to the client.

In the default mode of operation, CMAN-TDM creates one connection to the database for each incoming connection from the client. Idling client sessions can unnecessarily keep database connections engaged. To avoid the session idling, you can configure CMAN-TDM in the PRCP mode. In the PRCP mode, CMAN-TDM maintains an OCI session pool (OCISessionPool) and behaves like DRCP. When an application thread requires to interact with the database on a connection, CMAN-TDM picks up a session from the OCISessionPool and maps the incoming connection to the session. When the application thread indicates that it is done with the database activity, the connection is handed back (or unmapped) to the CMAN gateway process. The outgoing session is then released back to the OCISessionPool in CMAN-TDM. As a result, CMAN-TDM has fewer processes and sessions on the database than the connections.

Use the following connect descriptor in `tnsnames.ora` file to switch different connect modes for Implicit Connection Pooling. The connection string can point either to DRCP or to CMAN-TDM in the PRCP mode:

```
inst1s=(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=slc11xgx)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=t1.regress.rdbms.dev.us.oracle.com)(SERVER=POOLED)
(POOL_BOUNDARY=STATEMENT)))

inst1t=(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=slc11xgx)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=t1.regress.rdbms.dev.us.oracle.com)(SERVER=POOLED)
(POOL_BOUNDARY=TRANSACTION)))
```

In the above descriptors, the `CONNECT_DATA` section contains the following new attribute:

- For `inst1s`, the `POOL_BOUNDARY=STATEMENT` parameter specifies that implicit pooling semantics apply on a statement boundary basis.
- For `inst1t`, the `POOL_BOUNDARY=TRANSACTION` parameter specifies that implicit pooling semantics apply on a transaction boundary basis.

3.10.15.7 Setting or Resetting the Session State at the Boundaries During Deployment

If you have application administrator privileges with invoker rights, you must define a package called `ORA_CPOOL_STATE` once for every application user. Within the package, you can define a procedure called `ORA_CPOOL_STATE_CALLBACK` having two VARCHAR2 parameters, namely

`connection_class` and `service`. After the package is defined, any implicit get call on the server invokes the package and procedures defined in it.
Implicit Connection Pooling works seamlessly with connection class. If applications use connection class, then every implicit get call that happens on the server, CMAN-TDM, or both, honors the connection class setting.

Note:

If your application alters the session parameters that affect a query or DML (for example, while dealing with timestamp values) and you want to set the state as needed, you must move the `ALTER SESSION` logic to the PL/SQL callbacks. This is required because if your application runs an `ALTER SESSION` statement followed by the query execution, it might run on two different pooled servers with implicit pooling.

Alternatively, you can have the application run the `ALTER SESSION` statement and the queries together in an anonymous PL/SQL block.

Here is an example that uses only the `connection_class` and `service` parameters in its fixup logic.

```
CREATE OR REPLACE PACKAGE ORA_CPOOL_STATE authid current_user AS
PROCEDURE ORA_CPOOL_STATE_GET_CALLBACK(varchar2 in service, varchar2 in
connection_class);
PROCEDURE ORA_CPOOL_STATE_RLS_CALLBACK(varchar2 in service, varchar2 in
connection_class);
END;
/
CREATE OR REPLACE PACKAGE BODY ORA_CPOOL_STATE AS
PROCEDURE ORA_CPOOL_STATE_GET_CALLBACK(varchar2 in service, varchar2 in
connection_class)
IS
BEGIN
  IF (connection_class = 'GERMAN') THEN
    ALTER SESSION SET NLS_CURRENCY='€';
  ELSE IF (connection_class = 'INDIAN') THEN
    ALTER SESSION SET NLS_CURRENCY='₹';
  END IF;
END;

PROCEDURE ORA_CPOOL_STATE_RLS_CALLBACK(varchar2 in service, varchar2 in
connection_class)
IS
BEGIN
/* clear all the session state by restoring to defaults */
  IF (service = 'HR') THEN
    ALTER SESSION SET NLS_CURRENCY='$';
    ALTER SESSION SET date_format='';
END;
END;
```

3.10.15.8 Using the Session Cached Cursors with Implicit Connection Pooling

Implicit Connection Pooling clears the statement cache each time a session is implicitly released. As a result, multiple executions of the same query work as fully executed, separate queries instead of re-executed, same queries. Use session cached cursors to compensate for these shortcomings. Add the following to set session cached cursors in the `init.ora` parameter file:

```
session_cached_cursors=20
```

3.10.15.9 Security

For security, Implicit Connection Pooling ensures that a user session implicitly released to the DRCP or PRCP pool is available for requests only for the same user. If a current connected user has defined the `ORA_CPOOL_STATE` package and the `ORA_CPOOL_STATE_CALLBACK` callback, the package and the callback execute only in the context of the current connected user. A connected user, say Scott, cannot execute a callback of another user, say HR.

Related Topics

- *Oracle Call Interface Programmer's Guide*
- *Oracle C++ Call Interface Programmer's Guide*
- *Oracle Database JDBC Developer’s Guide*
- *Oracle Universal Connection Pool Developer’s Guide*
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows*

3.10.16 Benefiting from the Scalability of DRCP in an OCI Application

Consider the following OCI application scenarios and how they benefit from DRCP:

- An application neither uses the OCI session pool nor specifies a connection class or purity setting (or specifies `PURITY=NEW`).

  The application gets a new session from DRCP. When the application returns a connection to the pool, the session is not shared with other instances of the same application by default. Therefore, the pooled server remains assigned to the client for the life of the client session. (SQL*Plus is an example of a client that does not use the OCI session pool. SQL*Plus keeps connections even when they are idle.)

  The application benefits from reusing an existing pooled server.
- An application invokes `OCISessionGet()` outside of the OCI session pool, or to specify the connection class and `PURITY=SELF`.

  The application can reuse both DRCP pooled servers and sessions. However, after an `OCISessionRelease()` call, OCI terminates the connection to the connection broker. On the next `OCISessionGet()` call, the application reconnects to the broker, and then DRCP assigns a pooled server (and session) belonging to the specified connection class. Reconnecting incurs the cost of connection establishment and reauthentication.

  The application achieves better sharing of DRCP resources (processes and sessions) but does not benefit from caching connections to the connection broker.
- An application uses OCI session pool APIs, specifies a connection class, and specifies `PURITY=SELF`.

The application uses all DRCP functionality, reusing both the pooled server and the associated session and benefiting from cached connections to the connection broker. Cached connections do not incur the cost of reauthentication on the `OCISessionGet()` call.

See Also:

`OCISessionPoolCreate()`

`OCISessionGet()`

`OCISessionRelease()`,

`OCISessionPoolDestroy()`

3.10.17 Benefiting from the Scalability of DRCP in a Java Application

A customer who uses Universal Connection Pool (UCP), or uses ConnectionPoolDataSource as the connection factory, can upgrade to using DRCP by changing only the configuration (not the code).

See Also:

- Benefiting from the Scalability of DRCP in an OCI Application for more information about how Java applications benefit from DRCP as OCI applications

3.10.18 Best Practices for Using DRCP

The steps for designing an application that can benefit from the full power of DRCP are very similar to those for an application that uses the OCI session pool.

The only additional step is that for best performance, when deployed to run with DRCP, the application must explicitly specify a connection class.

Multiple instances of the same application must specify the same connection class for best performance and enhanced sharing of DRCP resources. Ensure that the different instances of the application can share database sessions.

Example 3-8 shows a DRCP application.

See Also:

- *Oracle Call Interface Programmer's Guide*

Example 3-8 DRCP Application

```
/* Assume that all necessary handles are allocated. */

/*   This middle tier uses a single database user. Create a homogeneous
     client-side session pool */
```

```
OCISessionPoolCreate (envhp, errhp, spoolhp, &poolName, &poolNameLen, "BOOKSDB",
    strlen("BOOKSDB"), 0, 10, 1, "SCOTT", strlen("SCOTT"), "password",
    strlen("password"), OCI_SPC_HOMOGENEOUS);

while (1)
{
   /* Process a client request */
   WaitForClientRequest();
   /* Application function */

   /* Set the Connection Class on the OCIAuthInfo handle that is passed as
      argument to OCISessionGet*/

   OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO,  "BOOKSTORE", strlen("BOOKSTORE"),
               OCI_ATTR_CONNECTION_CLASS, errhp);

   /* Purity need not be set, as default is OCI_ATTR_PURITY_SELF
      for OCISessionPool connections */

   /* You can get a SCOTT session released by Middle-tier 2 */
   OCISessionGet(envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
                 NULL, NULL, NULL, OCI_SESSGET_SPOOL);

   /* Database calls using the svchp obtained above  */
   OCIStmtExecute(...)

   /* This releases the pooled server on the database for reuse */
   OCISessionRelease (svchp, errhp, NULL, 0, OCI_DEFAULT);
}

/* Middle tier is done - exiting */
OCISessionPoolDestroy (spoolhp, errhp, OCI_DEFAULT);
```

Example 3-9 and Example 3-10 show connect strings that deploy code in 10 middle-tier hosts that service the `BOOKSTORE` application from Example 3-8.

In Example 3-9, assume that the database is Oracle Database 12*c* (or earlier) in dedicated server mode with DRCP not enabled and that the client has 12*c* libraries. The application gets dedicated server connections from the database.

Example 3-9   Connect String for Deployment in Dedicated Server Mode Without DRCP

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oraclehost.company.com)
 (PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)))
```

In Example 3-10, assume that DRCP is enabled on the Oracle Database 12*c* database. All middle-tier processes can benefit from the pooling capability of DRCP. The database resource requirement with DRCP is much less than it would be in dedicated server mode.

Example 3-10   Connect String for Deployment With DRCP

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oraclehost.company.com)
 (PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)(SERVER=POOLED)))
```

3.10.19 Compatibility and Migration

An OCI application linked with Oracle Database 12*c* client libraries works unaltered with:

- An Oracle Database 12*c* database with DRCP disabled
- A database server from a release earlier than Oracle Database 12*c*

- An Oracle Database 12*c* database server with DRCP enabled, when deployed with the DRCP connect string

Suitable clients benefit from enhanced scalability offered by DRCP if they are appropriately modified to use the OCI session pool APIs with the connection class and purity settings previously described.

See Also:

- *Oracle Database JDBC Developer's Guide* for more information about Oracle JDBC drivers support for DRCP

3.10.20 Using DRCP with Oracle Database Native Network Encryption

You can use the following supported checksum algorithms and encryption algorithms while using DRCP with Oracle Database Native Network Encryption.

Checksum Algorithms

- SHA1
- SHA256
- SHA384
- SHA512

Encryption Algorithms

- AES128
- AES192
- AES256

The DES, DES40, 3DES112, 3DES168, MD5, RC4_40, RC4_56, RC4_128, and RC4_256 algorithms are deprecated in this release. To transition your Oracle Database environment to use stronger algorithms, download and install the patch described in My Oracle Support note 2118136.2.

3.10.21 DRCP Restrictions

The following cannot be performed when connected to a DRCP pooled server:

- Shutting down the database
- Stopping DRCP
- Changing the password for the connected user
- Using shared database links to connect to a DRCP that is on a different instance
- Using TCPS or IPC connections
- Using enterprise user security with DRCP
- Using migratable sessions on the server side, either directly (using the `OCI_MIGRATE` option) or indirectly (invoking `OCIConnectionPoolCreate()`)

- Creating multiple sessions on a DRCP server for session switching or for dual session proxy
- Using initial client roles
- Using application context attributes (such as `OCI_ATTR_APPCTX_NAME` and `OCI_ATTR_APPCTX_VALUE`)

Attempting any of the aforementioned may result in the following error: ORA-56609: Usage not supported with DRCP.

Sessions created before DDL statements run can be assigned to clients after DDL statements run. Therefore, be careful when running DDL statements that affect database users in the pool. For example, before dropping a user, ensure that there are no sessions of that user in the pool and no connections to the broker that were authenticated as that user.

If sessions with explicit roles enabled are released to the pool, they can later be assigned to connections (of the same user) that need the default logon role. Therefore, avoid releasing sessions with explicit roles; instead, terminate them.

You can use Application Continuity with DRCP but you must ensure that the sessions are returned to the pool with the default session state.

Note:

You can use Oracle Advanced Security features such as encryption and strong authentication with DRCP.

Users can mix data encryption/data integrity combinations. However, users must segregate each such combination by using connection classes. For example, if the user application must specify AES256 as the encryption mechanism for one set of connections and AES128 for another set of connections, then the application must specify different connection classes for each set.

3.10.22 Using DRCP with Custom Pools

Oracle highly recommends using the OCI session pool, which is already integrated with DRCP, FAN, and RLB.

However, an application that does not use the OCI session pool can still use DRCP if either of the following is true:

- The application was built using its own custom connection pool.
- The application uses no pool, but has periods when it does not use its session (and could therefore release it to a pool) and does not depend on getting back the same session

To use DRCP with such an application, the session must be stateful; that is, the session must have the `OCI_ATTR_SESSION_STATE` attribute. When an application is stateful and DRCP is enabled, OCI transparently assigns it an appropriate session from the DRCP pool. If the application is stateless (has the `OCI_SESSION_STATELESS` attribute) and DRCP is enabled, OCI transparently returns the session to the DRCP pool.

Applications must identify session state as promptly as possible for efficient utilization of underlying database resources.

Note:

An application that specifies the attribute `OCI_ATTR_SESSION_STATE` or `OCI_SESSION_STATELESS` must also specify session purity and connection class.

See Also:

- *Oracle Call Interface Programmer's Guide* for more information about `OCI_ATTR_SESSION_STATE` attribute

3.10.23 Explicitly Marking Sessions Stateful or Stateless

An application typically requires a specific database session for the duration of a unit of work. For this duration, the session is **stateful**. After this duration, if the application does not depend on retaining the specific session for subsequent units of work, then the session is **stateless**.

When an application or caller detects a session's transition from stateful to stateless, or the reverse, the application can explicitly inform OCI of the transition by using the `OCI_ATTR_SESSION_STATE` or `OCI_SESSION_STATELESS` attribute. This information lets OCI and Oracle Database transparently perform scalability optimizations, such as reassigning the session that the application is not using to someone else and then assigning the application a new session when necessary.

See Also:

Using DRCP with Custom Pools

Example 3-11 shows a code fragment that explicitly marks session states.

Example 3-11 Explicitly Marking Sessions Stateful or Stateless

```
wait_for_transaction_request();
do {

ub1 state;

/* mark  database session as STATEFUL  */
state = OCI_SESSION_STATEFUL;
checkerr(errhp, OCIAttrSet(usrhp, OCI_HTYPE_SESSION,
        &state, 0, OCI_ATTR_SESSION_STATE, errhp));
/* do database work consisting of one or more related calls to the database */

...

/* done with database work, mark session as stateless */
state = OCI_SESSION_STATELESS;
checkerr(errhp, OCIAttrSet(usrhp, OCI_HTYPE_SESSION,
         &state, 0,OCI_ATTR_SESSION_STATE, errhp));
wait_for_transaction_request();

} while(not _done);
```

A session obtained from outside the OCI session pool is marked `OCI_SESSION_STATEFUL` and remains `OCI_SESSION_STATEFUL` unless the application explicitly marks it `OCI_SESSION_STATELESS`.

A session obtained from the OCI session pool is marked `OCI_SESSION_STATEFUL` by default when the first call is initiated on that session. When the session is released to the pool, it is marked `OCI_SESSION_STATELESS` by default. Therefore, you need not explicitly mark sessions as stateful or stateless when you use the OCI session pool.

See Also:

Oracle Call Interface Programmer's Guide for more information about `OCI_ATTR_SESSION_STATE`

3.10.24 Using DRCP with Oracle Real Application Clusters

Oracle Real Application Clusters (Oracle RAC) is a database option in which a single database is hosted by multiple instances on multiple nodes. When DRCP is configured in a database in an Oracle RAC environment, the pool configuration is applied to each database instance. Starting or stopping the pool on one instance starts or stops the pool on all instances.

3.10.25 DRCP with Data Guard

When operating DRCP in a Data Guard environment:

- On a physical standby database:
  - You can start the pool only if the pool is running on the primary database.
  - You can stop the pool only if the pool is stopped on the primary database.
  - You cannot configure, restore to defaults, or alter pool parameters.

  The preceding restrictions cease to apply to the physical standby database if it becomes the primary database.
- On a logical standby database, all pool operations are allowed.

3.11 Memoptimize Pool

This pool optimizes the read operation for select statements

Memoptimize Rowstore performs high-performance reads for tables specified with the `MEMOPTIMIZE FOR READ` clause.

Note:

Deferred inserts cannot be rolled back because they do not use standard locking and redo mechanisms

Related Topics

- *Oracle Database Concepts*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Reference*

3.12 Oracle RAC Sharding

This section explains about Oracle RAC Sharding

Oracle RAC Sharding increases the performance and scalability of a Oracle RAC database with minimal application changes. It affinitizes table partitions to Oracle RAC instances, and routes the database requests, which specify a partitioning key to the instance that logically holds the corresponding partition. This provides better cache utilization and reduces block pings across instances. The partitioning key can be added only to the most performance critical requests. Requests that do not specify the key works transparently and can be routed to any instance. This feature can be enabled by executing an `ALTER SYSTEM` command and without modifying the database schema and SQL statements.

Note:

The partitioning key value must be provided when requesting a database connection.

Related Topics

- *Oracle Database Net Services Administrator's Guide*
- *Oracle Call Interface Programmer's Guide*
- *Oracle Universal Connection Pool Developer's Guide*
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows*
- *Oracle Real Application Clusters Administration and Deployment Guide*
- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*