5 Optimizing Universal Connection Pool Behavior

This chapter describes the following concepts:

5.1 Optimizing Connection Pools

This section provides instructions for setting connection pool properties in order to optimize pooling behavior. Upon creation, UCP JDBC connection pools are pre-configured with a default setup. The default setup provides a general, all-purpose connection pool. However, different applications may have different database connection requirements and may want to modify the default behavior of the connection pool. Behaviors, such as pool size and connection timeouts can be configured and can improve overall connection pool performance as well as connection availability. In many cases, the best way to tune a connection pool for a specific application is to try different property combinations using different values until optimal performance and throughput is achieved.

Setting Connection Pool Properties

Connection pool properties are set either when getting a connection through a pool-enabled data source or when creating a connection pool using the connection pool manager.

The following example demonstrates setting connection pool properties though a pool-enabled data source:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("JDBC_UCP");
pds.setMinPoolSize(4);pds.setMaxPoolSize(20);
...
```

The following example demonstrates setting connection pool properties when creating a connection pool using the connection pool manager:

```
UniversalConnectionPoolManager mgr = UniversalConnectionPoolManagerImpl.
getUniversalConnectionPoolManager();

pds.setConnectionPoolName("JDBC_UCP");
pds.setMinPoolSize(4);pds.setMaxPoolSize(20);
...

mgr.createConnectionPool(pds);
```

5.2 About Controlling the Pool Size in UCP

UCP JDBC connection pools include a set of properties that are used to control the size of the pool. The properties allow the number of connections in the pool to increase and decrease as demand increases and decreases. This dynamic behavior helps conserve system resources that are otherwise lost on maintaining unnecessary connections.

This section describes the following topics:

- Setting the Initial Pool Size
- Setting the Minimum Pool Size
- Setting the Maximum Pool Size
- Setting the Minimum Idle Connection Number

5.2.1 Setting the Initial Pool Size

The initial pool size property specifies the number of available connections that are created when the connection pool is initially created or re-initialized. This property is typically used to reduce the ramp-up time incurred by priming the pool to its optimal size.

A value of `0` indicates that no connections are pre-created. The default value is `0`. The following example demonstrates configuring an initial pool size:

```
pds.setInitialPoolSize(5);
```

If the initial pool size property is greater than the maximum pool size property, then only the maximum number of connections are initialized.

If the initial pool size property is less than the minimum pool size property, then only the initial number of connections are initialized and maintained until enough connections are created to meet the minimum pool size value.

If during the pool initialization process, connections cannot be created up to the value specified in the `initPoolSize` property, then the pool attempts to create the remaining connections to fulfill the initial pool size. If the pool does not have the physical ability to do so, then the pool initialization process ends and it keeps trying to maintain the designated minimum and maximum connection limits for the rest of its life cycle.

5.2.2 Setting the Minimum Pool Size

The minimum pool size property specifies the minimum amount of available connections and borrowed connections that a pool maintains. A connection pool always tries to return to the minimum pool size specified unless the minimum amount is yet to be reached. For example, if the minimum limit is set to `10` and only 2 connections are ever created and borrowed, then the number of connections maintained by the pool remains at `2` because this number is less than the minimum pool size.

This property allows the number of connections in the pool to decrease as demand decreases. At the same time, the property ensures that system resources are not wasted on maintaining connections that are unnecessary.

The default value is `0`. The following example demonstrates configuring a minimum pool size:

```
pds.setMinPoolSize(2);
```

5.2.3 Setting the Maximum Pool Size

The maximum pool size property specifies the maximum number of available and borrowed (in use) connections that a pool maintains. If the maximum number of connections are borrowed, no connections will be available until a connection is returned to the pool.

This property allows the number of connections in the pool to increase as demand increases. At the same time, the property ensures that the pool does not grow to the point of exhausting the resources of a system, which ultimately affects the performance and availability of an application.

A value of `0` indicates that no connections are maintained by the pool. An attempt to get a connection results in an exception. The default value is to allow the pool to continue to create connections up to `Integer.MAX_VALUE` (2147483647 by default). The following example demonstrates configuring a maximum pool size:

```
pds.setMaxPoolSize(100);
```

5.2.4 Setting the Minimum Idle Connection Number

The minimum idle connection number property specifies the minimum number of idle connections that the connection pool maintains.

If the number of available connections is less than the number of minimum idle connections specified, then new connections are created in the background and made available in the pool. The range of valid values for this property ranges from 0 to `Integer.MAX_VALUE`. The default value is 0. It is illegal to set this property to a value greater than the maximum pool size.

The following code snippet shows how to use this property:

```
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("<URL>");
pds.setUser("<user_name>");
pds.setPassword("<password>");
pds.setInitialPoolSize(5);
pds.setMinPoolSize(5);
pds.setMinIdle(5);
pds.setMaxPoolSize(10);
Connection conn = pds.getConnection();
conn.close();
```

See Also:

Oracle Universal Connection Pool Java API Reference for more information about this property

5.3 About Optimizing Real-World Performance with Static Connection Pools

Most on-line transaction processing (OLTP) performance problems that the Real-World Performance group investigates relate to the connection strategy used by the application. For this reason, designing a sound connection strategy is crucial for system performance, especially in enterprise environments that must scale to meet increasing demand.

Most applications use a dynamic pool of connections to the database, configured with a minimum number of connections to keep open on the database and a maximum number of connections that can be made to the database. When an application needs a connection to the database, then it requests one from the pool. If there are no connections available, then the application creates a new connection, if it has not reached the maximum number of connections already. If a connection has not been used for a specified duration of time, then the application closes the connection, if there are more than the minimum number of connections available.

This configuration conserves system resources by only maintaining the number of connections actively needed by the application. In the real world, this configuration enables connection storms and database system CPU oversubscription, quickly destabilizing a system. A connection storm can occur when there are lots of activities on the application server requiring database connections. If there are not enough connections to the database to serve all of the requests, then the application server opens new connections. Creating a new connection to the database is a resource intensive activity, and when lots of connections are made in a short period of time, it can overwhelm the CPU resources on the database system.

So, for creating a static connection pool, the number of connections to the database system must be based on the CPU cores available on the system. Oracle recommends 1-10 connections per CPU core. The ideal number varies depending on the application and the system hardware. However, the value is somewhere within that range. the Real-World Performance group recommends creating a static pool of connections to the database by setting the minimum and maximum number of connections to the same value. This prevents connection storms by keeping the number of database connections constant to a predefined value.

For example, if a database server has 2 CPUs, 12 cores per CPU, and 2 threads per core, then there are 24 cores available and the number of connections to the database should be between 24 and 240. The number of threads is not taken into consideration as only the CPU cores are able to execute instructions. This number is cumulative for all applications connecting to the system and for all databases, if there is more than one database on the system. If there are two application servers, then the maximum number of connections (for example, 240 in this case) should be divided between them. If there are two databases running on the system, then the maximum number of connections that is, 240 connections needs to be divided between them.

See Also:

- https://www.youtube.com/watch?v=Oo-tBpVewP4
- https://www.youtube.com/watch?v=XzN8Rp6glEo

5.4 Stale Connections in UCP

Stale connections are connections that remain either available or borrowed, but are no longer being used. Stale connections that remain borrowed may affect connection availability.

In addition, stale connections may impact system resources that are used to maintain unused connections for extended periods of time. The pool properties discussed in this section are used to control stale connections.

This section describes the following topics:

- What is Connection Reuse?
- Setting the Connection Validation Timeout
- Setting the Abandon Connection Timeout
- Setting the Time-To-Live Connection Timeout
- Setting the Connection Wait Timeout
- Setting the Inactive Connection Timeout
- Setting the Query Timeout
- Setting the Timeout Check Interval

Note:

It is good practice to close all connections that are no longer required by an application. Closing connections helps minimize the number of stale connections that remain borrowed.

5.4.1 What is Connection Reuse?

The connection reuse feature allows connections to be gracefully closed and removed from a connection pool after a specific amount of time or after the connection has been used a specific number of times. This feature saves system resources that are otherwise wasted on maintaining unusable connections.

5.4.1.1 Setting the Maximum Connection Reuse Time

The maximum connection reuse time allows connections to be gracefully closed and removed from the pool after a connection has been in use for a specific amount of time. The timer for this property starts when a connection is physically created. Borrowed connections are closed only after they are returned to the pool and the reuse time is exceeded.

This feature is typically used when a firewall exists between the pool tier and the database tier and is setup to block connections based on time restrictions. The blocked connections remain in the pool even though they are unusable. In such scenarios, the connection reuse time is set to a smaller value than the firewall timeout policy.

Note:

The maximum connection reuse time is different from the time-to-live connection timeout. The main difference between the maximum connection reuse time and the time-to-live is that time-to-live can cancel a connection, but maximum connection reuse time never does that. The maximum connection reuse time handler gets applied only in the following cases:

- Prior to a connection borrow
- Immediately after a connection is returned back to a pool
- On periodic basis, with every `timeoutCheckInterval`

The maximum connection reuse time value is represented in seconds. The default value is `0`, which indicates that this feature is disabled. The following example demonstrates configuring a maximum connection reuse time:

```
pds.setMaxConnectionReuseTime(300);
```

Starting from Oracle Database Release 23ai, you can use the new system property `oracle.ucp.timersAffectAllConnections` to change the behavior of the maximum connection reuse time processing. The default value of this property is `FALSE`, which means that during periodical appropriate time processing, a pool is scanned down to a minimum pool size and the pool never closes connections below the minimum pool size. If the `SYSTEM_PROPERTY_TIMERS_AFFECT_ALL_CONNECTIONS` system property is set to `TRUE`, then the periodic poll checks all available connection for the maximum connection reuse time criteria, so the pool size may go below the minimum pool size, replacing the applicable connections.

Related Topics

- Setting the Time-To-Live Connection Timeout

5.4.1.2 Setting the Maximum Connection Reuse Count

The maximum connection reuse count allows connections to be gracefully closed and removed from the connection pool after a connection has been borrowed a specific number of times. This property is typically used to periodically recycle connections in order to eliminate issues such as memory leaks.

A value of `0` indicates that this feature is disabled. The default value is `0`. The following example demonstrates configuring a maximum connection reuse count:

```
pds.setMaxConnectionReuseCount(100);
```

5.4.2 Setting the Connection Validation Timeout

The connection validation timeout specifies the duration within which a borrowed connection from the pool is validated. This is the maximum time for a connection validation operation. If the validation is not completed during this period, then the connection is treated as invalid.

The connection validation timeout value represents seconds. The default value is set to 15. The following example demonstrates configuring a connection validation timeout:

```
pd.setConnectionValidationTimeout(55);
```

5.4.3 Setting the Abandon Connection Timeout

The abandoned connection timeout (ACT) enables borrowed connections to be reclaimed back into the connection pool after a connection has not been used for a specific amount of time. Abandonment is determined by monitoring calls to the database.

The abandoned connection timeout feature helps maximize connection reuse and conserves system resources that are otherwise lost on maintaining borrowed connections that are no longer in use.

Note:

Before reclaiming connections for reuse, UCP either cancels or rolls back the connections that have local transactions pending.

The ACT value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `0`. The following example demonstrates configuring an abandoned connection timeout:

```
pds.setAbandonedConnectionTimeout(10);
```

Every connection is reaped after a specific period of time. Either it is reaped when ACT expires, or, if it is immune from ACT, then it is reaped after the immunity expires. If you set ACT on a pool, then the following connection reaping policies apply:

- If a statement is executed without calling the `Statement.setQueryTimeout` method on that statement, then the connection is reaped if ACT is exceeded, even though the connection is waiting for the server to respond to the query.
- If a statement is executed with calling the `Statement.setQueryTimeout` method, then the connection is reaped after the query timeout and ACT have expired. The connection is not reaped while waiting on the query timeout. The expiration of the query timeout is an event that resets the ACT timer. If the ACT expires while waiting for the `cancel` action that occurs at the expiration of the query time out, then the connection is reaped.
- The default query timeout in the UCP is set to zero (0) for an appropriate pool data source, using the `PoolDataSource.setQueryTimeout` method, if the ACT is set to 0. If the ACT is greater than zero (0), then the default query timeout is set to 60 seconds.
- If a connection has two statements: s1 with a query timeout and s2 without a query timeout, then ACT does not reap the connection while s1 waits for the query timeout, but reaps the connection if s2 hangs.

  Note that the two statements execute sequentially based on JDBC requirement.

5.4.4 Setting the Time-To-Live Connection Timeout

The time-to-live connection timeout enables borrowed connections to remain borrowed for a specific amount of time before the connection is reclaimed by the pool. This timeout feature helps maximize connection reuse and helps conserve systems resources that are otherwise lost on maintaining connections longer than their expected usage.

Note:

UCP either cancels or rolls back connections that have local transactions pending before reclaiming connections for reuse.

The time-to-live connection timeout value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `0`. The following example demonstrates configuring a time-to-live connection timeout:

```
pds.setTimeToLiveConnectionTimeout(18000)
```

5.4.5 Setting the Connection Wait Timeout

The connection wait timeout specifies how long an application request waits to obtain a connection if there are no longer any connections in the pool. A connection pool runs out of connections if all connections in the pool are being used (borrowed) and if the pool size has reached it's maximum connection capacity as specified by the maximum pool size property. The request receives an SQL exception if the timeout value is reached. The application can then retry getting a connection. This timeout feature improves overall application usability by minimizing the amount of time an application is blocked and provides the ability to implement a graceful recovery.

The connection wait timeout value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `3` seconds. The following example demonstrates configuring a connection wait timeout:

```
pds.setConnectionWaitTimeout(10);
```

5.4.6 Setting the Inactive Connection Timeout

The inactive connection timeout specifies how long an available connection can remain idle before it is closed and removed from the pool.

This timeout property is only applicable to available connections and does not affect borrowed connections. This property helps conserve resources that are otherwise lost on maintaining connections that are no longer being used. The inactive connection timeout (together with the maximum pool size) enables a connection pool to grow and shrink as application load changes.

The inactive connection timeout value is represented in seconds. A value of `0` indicates that the feature is disabled. The default value is set to `0`. The following example demonstrates configuring an inactive connection timeout:

```
pds.setInactiveConnectionTimeout(60);
```

Starting from Oracle Database Release 23ai, you can use the new system property `oracle.ucp.timersAffectAllConnections` to change the behavior of the Inactive Connection Timeout and Maximum Connection Reuse Time properties. If you set this system property to `TRUE`, then the periodic poll checks all the available connections for the maximum connection reuse time and inactive connection timeout criteria, and closes all the connections that satisfy the criteria. This may make the pool size go below the minimum pool size, resulting in the creation of new connections by the pool to maintain the minimum pool size limit.

See Also:

Setting the Maximum Connection Reuse Time for more information about the `oracle.ucp.timersAffectAllConnections` property

5.4.7 Setting the Query Timeout

In Oracle Database 12*c* Release 2 (12.2.0.1), UCP introduced the `queryTimeout` property. This property specifies the number of seconds UCP waits for a `Statement` object to execute. If the limit is exceeded, then a `DatabaseException` is thrown. Use the `setQueryTimeout` method for setting this property in the following way:

```
...
PoolDataSourceImpl pds = new PoolDataSourceImpl();
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL(<url>);
pds.setUser("scott");
pds.setPassword(<password>);
pds.setConnectionPoolName("my_pool");
pds.setQueryTimeout(60); // 60 seconds to wait on query
...
```

5.4.8 Setting the Timeout Check Interval

The timeout check interval property controls how frequently the timeout properties (abandoned connection timeout, time-to-live connection timeout, and inactive connection timeout) are enforced. Connections that have timed-out are reclaimed when the timeout check cycle runs. This means that a connection may not actually be reclaimed by the pool at the moment that the connection times-out. The lag time between the connection timeout and actually reclaiming the connection may be considerable depending on the size of the timeout check interval.

The timeout check interval property represents seconds. The default value is set to `30`. The following example demonstrates configuring a property check interval:

```
pds.setTimeoutCheckInterval(60);
```

See Also:

Oracle Database Net Services Administrator's Guide for more information about Oracle Net Services

5.5 About Harvesting Connections in UCP

The connection harvesting feature allows a specified number of borrowed connections to be reclaimed when the connection pool reaches a specified number of available connections. This section describes the following concepts:

- Overview of Harvesting Connections in UCP

- Setting a Connection to Harvestable
- Setting the Harvest Trigger Count
- Setting the Harvest Maximum Count

5.5.1 Overview of Harvesting Connections in UCP

This feature helps ensure that a certain number of connections are always available in the pool and helps maximize performance. The feature is particularly useful if an application caches connection handles. Caching is typically performed for performance reasons because it minimizes re-initialization of state necessary for connections to participate in a transaction.

For example, a connection is borrowed from the pool, initialized with necessary session state, and then held in a context object. Holding connections in this manner may cause the connection pool to run out of available connections. The connection harvest feature reclaims the borrowed connections, if appropriate, and allows the connections to be reused.

Connection harvesting is controlled using the `HarvestableConnection` interface and configured or enabled using two pool properties: Connection Harvest Trigger Count and Connection Harvest Maximum Count. The interface and properties are used together when implementing the connection harvest feature.

5.5.2 Setting a Connection to Harvestable

The `setConnectionHarvestable(boolean)` method of the `oracle.ucp.jdbc.HarvestableConnection` interface controls whether or not a connection will be harvested. This method is used as a locking mechanism when connection harvesting is enabled. For example, the method is set to `false` on a connection when the connection is being used within a transaction and must not be harvested. After the transaction completes, the method is set to `true` on the connection and the connection can be harvested if required.

Note:

All connections are harvestable, by default, when the connection harvest feature is enabled. If the feature is enabled, the `setConnectionHarvestable` method should always be used to explicitly control whether a connection is harvestable.

The following example demonstrates using the `setConnectionHarvestable` method to indicate that a connection is not harvestable when the connection harvest feature attempts to harvest connections:

```
Connection conn = pds.getConnection();

((HarvestableConnection) conn).setConnectionHarvestable(false);
```

5.5.3 Setting the Harvest Trigger Count

The connection harvest trigger count specifies the available connection threshold that triggers connection harvesting. For example, if the connection harvest trigger count is set to 10, then connection harvesting is triggered when the number of available connections in the pool drops to 10.

A value of `Integer.MAX_VALUE` (2147483647 by default) indicates that connection harvesting is disabled. The default value is `Integer.MAX_VALUE`.

The following example demonstrates enabling connection harvesting by configuring a connection harvest trigger count.

```
pds.setConnectionHarvestTriggerCount(2);
```

5.5.4 Setting the Harvest Maximum Count

The connection harvest maximum count property specifies how many borrowed connections should be returned to the pool once the harvest trigger count has been reached. The number of connections actually harvested may be anywhere from 0 to the connection harvest maximum count value. Least recently used connections are harvested first which allows very active user sessions to keep their connections the most.

The harvest maximum count value can range from `0` to the maximum connection property value. The default value is `1`. An SQLException is thrown if an out-of-range value is specified.

The following example demonstrates configuring a connection harvest maximum count.

```
pds.setConnectionHarvestMaxCount(5);
```

Note:

- If connection harvesting and abandoned connection timeout features are enabled at the same time, then the timeout processing does not reclaim the connections that are designated as nonharvestable.
- If connection harvesting and time-to-live connection timeout features are enabled at the same time, then the timeout processing reclaims the connections that are designated as nonharvestable.

Related Topics

- Controlling Reclaimable Connection Behavior

5.6 About Caching SQL Statements in UCP

This section describes how to cache SQL statements in UCP, in the following sections:

- Overview of Statement Caching in UCP
- Enabling Statement Caching in UCP

5.6.1 Overview of Statement Caching in UCP

Statement caching makes working with statements more efficient. Statement caching improves performance by caching executable statements that are used repeatedly and makes it unnecessary for programmers to explicitly reuse prepared statements. Statement caching eliminates overhead due to repeated cursor creation, repeated statement parsing and creation and reduces overhead of communication between applications and the database. Statement caching and reuse is transparent to an application. Each statement cache is associated with a physical connection. That is, each physical connection will have its own statement cache.

The match criteria for cached statements are as follows:

- The SQL string in the statement must be the same (case-sensitive) to one in the cache.
- The statement type must be the same (`prepared` or `callable`) to the one in the cache.
- The scrollable type of result sets produced by the statement must be the same (`forward-only` or `scrollable`) as the one in the cache.

Statement caching is implemented and enabled differently depending on the JDBC driver vendor. The instructions in this section are specific to Oracle's JDBC driver. Statement caching on other vendors' drivers can be configured by setting a connection property on a connection factory. Refer to the JDBC vendor's documentation to determine whether statement caching is supported and if it can be set as a connection property. UCP does support JDBC 4.0 (JDK16) APIs to enable statement pooling if a JDBC vendor supports it.

Related Topics

- Setting Connection Properties

5.6.2 Enabling Statement Caching in UCP

The maximum number of statements property specifies the number of statements to cache for each connection. The property only applies to the Oracle JDBC driver. If the property is not set, or if it is set to `0`, then statement caching is disabled. By default, statement caching is disabled. When statement caching is enabled, a statement cache is associated with each physical connection maintained by the connection pool. A single statement cache is not shared across all physical connections.

The following example demonstrates enabling statement caching:

```
pds.setMaxStatements(10);
```

Determining the Statement Cache Size

The cache size should be set to the number of distinct statements the application issues to the database. If the number of statements that an application issues to the database is unknown, use the JDBC performance metrics to assist with determining the statement cache size.

Statement Cache Size Resource Issues

Each connection is associated with its own statement cache. Statements held in a connection's statement cache may hold on to database resources. It is possible that the number of opened connections combined with the number of cached statements for each connection could exceed the limit of open cursors allowed for the database. This issue may be avoided by reducing the number of statements allowed in the cache, or by increasing the limit of open cursors allowed by the database.

5.7 UCP Best Practices

Universal Connection Pool has an extensive collection of tools and APIs to analyze connection leaks and tune up pool properties for optimizing its operation. This section describes these tools and APIs.

The `All connections in the Universal Connection Pool are in use` exception indicates the shortage of connections in the pool at a given time, which means that the pool is unable to meet the connection borrowing requests of an application. This can happen due to the following reasons:

- An application borrows connections and holds them for a long time without usage, never returning them to the pool. Connection leakage can also happen when an application borrows connections, holds them without usage, and finally returns them to the pool after a very long time. These are the classical connection leakage use cases. You must eliminate non-productive connection borrowings that last for long or infinite periods of time.
- The pool has insufficient capacity for processing the whole flow of connection borrowing requests. In this case, the connection supply of the pool is not enough to perform the expected job and this results in the exception. You must increase the pool capacity in such a case.

Following is a list of useful tools and APIs that you must be aware of prior to debugging and tuning up UCP:

- **Abandoned Connection Timeout (ACT)**: This API enables setting up a timeout for a connection that is borrowed but unused.

  > **See Also:**
  >
  > Setting the Abandon Connection Timeout

- **Time-To-Live Connection Timeout (TTL)**: This API too enables setting up a timeout for a connection that is borrowed but unused. However, it also furnishes information about the borrowed connections that are busy with associated on-going processes. It also enables to reclaim these busy connections back to a pool and restores the capacity of the pool, in case of very long processes.

  > **See Also:**
  >
  > Setting the Time-To-Live Connection Timeout

- **Connection Harvesting Mechanism**: This is a special API that enables UCP to always keep certain number of connections available for borrowing and in turn, helps in avoiding the `All connections in the Universal Connection Pool are in use` exception.

  > **See Also:**
  >
  > About Harvesting Connections in UCP

- **Connection Wait Timeout (CWT)**: This is an important property when you try to tune up UCP to avoid the `All connections in the Universal Connection Pool are in use` exception. When an application attempts to borrow a connection out of a pool and there are no available connections at that time, UCP waits for an available connection to appear for the amount of time that is equal to the value of CWT. By default, CWT is set for 3 seconds. In many applications, you can increase this timeout to enable a pool to wait for longer for an available connection to appear, without getting the `All connections in the Universal Connection Pool are in use` exception.

  > **See Also:**
  >
  > Setting the Connection Wait Timeout

- **Maximum Pool Size (`MaxPoolSize`)**: This property affects the pool capacity and helps to avoid the `All connections in the Universal Connection Pool are in use` exception. Oracle recommends to have a small pool size, typically a small number multiplied by the number of cores on a database server. It is better to increase the CWT than making `MaxPoolSize` very high.

  > **See Also:**
  >
  > Setting the Maximum Pool Size

- **Inactive Connection Timeout (ICT) in combination with MaxPoolSize**: ICT is the timeout property that enables UCP to automatically close available connections that did not have a chance to be borrowed for a particular amount of time, which is specified by the value of ICT. This way, the UCP can avoid connections if the working set of the pool is too big to perform a given throughput. For the pool to auto-tune the required number of connections in the working set of the pool, you can set the `MaxPoolSize` parameter to a big value and set the ICT accordingly.

  > **Note:**
  >
  > Setting the Inactive Connection Timeout

- **Pool Size Auto Tuner**: This tool enables UCP to automatically tune up pool size for better throughput.
- **Pool Statistical Metrics**: This is a set of statistics that helps to determine the activities and statistics of a pool, for example, the number of available connections, the number of borrowed connections, and the Average Connection Wait Time (ACWT). ACWT can find a proper value of CWT property for pool tuning. If ACWT is big, then it indicates that the UCP is close to over-using its capacity.

  > **See Also:**
  >
  > Pool Statistics

- **Pool Logging**: UCP has an extensive and flexible logging system. Logging enables you to determine events related to connection opening, connection closing, connection borrowing, and wait time of return and borrow requests.

  > **See Also:**
  >
  > Overview of Logging and Tracing in UCP