DigiTalk

Oracle WebLogic Server

Tuning Data Source Connection Pools

A Comprehensive Guide to Oracle WebLogic Server JDBC Connection Pool Tuning

Based on: Administering JDBC Data Sources for Oracle WebLogic Server, Release 15.1.1

14.1.2 & 15.1.1
DT246055-03
July 2026

DigiTalk

Table of Contents

DigiTalk

1. Introduction

Connection pooling is one of the most important performance levers available to an Oracle WebLogic Server administrator. A JDBC data source's connection pool controls how physical database connections are created, cached, tested, retried, and recycled on behalf of applications. Tuned correctly, a connection pool minimizes database round-trips, avoids costly connection churn, protects the server from cascading failures during outages, and keeps latency predictable under load. Tuned poorly, it can exhaust database cursors, hide dead connections from applications, or bring an entire cluster to a crawl during a database blip.

This document consolidates Oracle's official guidance on tuning JDBC data source connection pools in WebLogic Server 15.1.1 into a single, organized reference. It covers statement caching, initial capacity handling, connection testing, retry and wait behavior, leaked-connection recovery, capacity planning, statement timeouts, thread pinning, object wrapping, background maintenance timers, and connection creation rate limiting.

2. Increasing Performance with the Statement Cache

Preparing and calling statements involves significant processing overhead, both in the network round-trip between the application server and the database and in the database engine itself. WebLogic Server reduces this overhead by caching prepared and callable statements so they can be reused instead of re-prepared on every call.

Each physical connection in a data source maintains its own statement cache, but the cache behavior (type and size) is configured once, at the data source level, and applied uniformly to every connection's cache.

> ***Note:*** *When using the Oracle JDBC driver, the Oracle driver's own prepared-statement cache should be used instead of the WebLogic Server data source statement cache.*

2.1 Statement Cache Algorithms

The Statement Cache Type determines which statements are kept in each connection's cache once it fills up.

LRU (Least Recently Used) — default

WebLogic Server caches statements until the cache reaches its configured size. When a new statement is prepared and the cache is full, the least recently used cached statement is evicted and replaced. WebLogic Server's LRU implementation uses an approximate LRU scheme rather than a strict one.

Fixed

WebLogic Server caches statements until the cache is full, then stops caching new statements entirely — additional statements are executed uncached. This can waste cache slots on rarely used statements. LRU is generally preferred because infrequently used statements naturally get evicted in favor of frequently used ones.

2.2 Statement Cache Size

Statement Cache Size sets how many prepared/callable statements are cached per connection, per data source instance. Because each cached statement typically consumes a database-side resource (such as a cursor), cache size has direct capacity implications:

- Resource cost per statement: a data source with 10 connections deployed on 2 managed servers, at the default cache size of 10, could open up to 200 cursors (10 connections × 2 servers × 10 cached statements) on the database.
- Memory cost: some drivers impose significant memory overhead per open statement; total memory scales with (number of data sources) × (number of connections) × (number of cached statements).
- Some DBMSs cap the number of statements or cursors allowed per connection.

The ideal cache size is application-dependent — ideally large enough to hold every distinct prepared/callable statement your application issues on a connection. A practical approach: temporarily set the cache size very large, observe pool statistics under real traffic, then set the size slightly above the largest observed value. There is no WebLogic-side performance penalty for over-sizing the cache.

Conversely, an undersized cache can hurt performance: high turnover flushes statements before they are ever reused. In extreme cases, where the application's statement diversity is too large to cache effectively, disabling the statement cache altogether may perform better than a too-small cache.

2.3 Usage Restrictions for the Statement Cache

Statement caching offers a large performance win but has real limitations. If you see errors related to prepared or callable statements, try setting the statement cache size to 0 (disabling caching) to determine whether caching is the cause.

DDL changes can break cached statements

A cached prepared statement refers to specific database objects and column data types as they existed when it was prepared. Dropping and recreating a referenced table, or adding/removing/reordering columns, can cause a previously working cached statement to fail on its next execution. Behavior here depends on the DBMS.

Using setNull correctly

When a cached prepared statement binds a null value with setNull, the exact SQL data type must be specified — a generic type can cause truncation or failure when the statement later runs with a non-null value.

```
java.sql.Types.Long sal=null
. . .
if (sal == null)
  setNull(2,int)   // INCORRECT
else
  setLong(2,sal)
```

The corrected version:

```
if (sal == null)
  setNull(2,long)  // CORRECT
else
  setLong(2,sal)
```

Cached statements may reserve database cursors

Every cached statement can hold open a database cursor. Caching too many statements can exceed a connection's cursor limit; mitigate this by raising the DBMS cursor limit or lowering the statement cache size.

Other considerations

- Setting the Oracle connection property oracle.jdbc.implicitstatementcachesize automatically forces the WebLogic statement cache size to 0.
- Data sources using Database Resident Connection Pooling (DRCP) clear the statement cache whenever the application closes the connection.
- Data sources using the JDBC Replay Driver automatically have their statement cache size forced to 0.
- Calling the JDBC 4.0 setPoolable(false) method removes a statement from the cache in addition to calling the method on the underlying driver object.

3. Initial Capacity Enhancement in the Connection Pool

Since WebLogic Server 12.2.1.3, three features improve how a data source builds its Initial Capacity connections when a server starts: connection retry, early failure handling, and critical data source marking. Prior to this release, the server attempted to create all initial-capacity connections even when some attempts failed, which could significantly slow startup if the network or database was unavailable.

3.1 Connection Retry

Two connection properties govern retrying failed initial connection creation:

- weblogic.jdbc.startupRetryCount — number of retry attempts after a failure; a value greater than 0 enables retry. Default: 0 (no retry).
- weblogic.jdbc.startupRetryDelaySeconds — delay, in seconds, between retries, used only when retry is enabled. Default: 0 (no delay).

3.2 Early Failure

The following property controls whether the data source keeps attempting to create the remaining initial-capacity connections after one attempt fails:

- weblogic.jdbc.continueMakeResourceAttemptsAfterFailure=true — if startup retry is enabled, this property is ignored, and the data source will not keep creating connections after a failure during server startup. It will, however, keep trying on a running server if the data source is deployed or redeployed while the server is up.

3.3 Critical Data Sources

If populating initial-capacity connections fails, by default the data source simply is not deployed (and so is unavailable via JNDI) while the server itself continues to start up normally and is not marked unhealthy. For applications where a data source is essential, this property changes that behavior:

- weblogic.jdbc.critical — when true, a managed server (not the administration server) fails to boot if this data source cannot be deployed. Default: false.

3.4 Example: WLST Configuration

The following WLST fragment configures retry count and delay on a data source:

```
edit()
startEdit()
datasource="dsname"
cd("/JDBCSystemResources/" + datasource + "/JDBCResource/" + datasource + "/JDBCDriverParams/"
+ datasource + "/Properties/" + datasource)
cmo.createProperty("weblogic.jdbc.startupRetryCount", "5")
cmo.createProperty("weblogic.jdbc.startupRetryDelaySeconds", "10")
save()
activate()
```

4. Connection Testing Options for a Data Source

To keep pooled connections healthy, WebLogic Server supports automatic testing (configured on the data source) and manual testing (used for troubleshooting). Automatic testing options are configured through the WebLogic Remote Console or WLST, via the JDBCConnectionPoolParamsBean.

4.1 Core Testing Attributes

- **Test Frequency** (TestFrequencySeconds) — seconds between background tests of idle connections; faulty connections found this way are closed and replaced. Requires Test Table Name to be set.
- **Test Reserved Connections** (TestConnectionsOnReserve) — tests each connection immediately before handing it to an application. Adds a small latency cost per request but guarantees the delivered connection is healthy. Also requires Test Table Name.
- **Test Table Name** (TestTableName) — the table (or, prefixed with SQL , a full SQL statement) used to validate a connection. Required to enable any connection testing at all.
- **Seconds to Trust an Idle Pool Connection** (SecondsToTrustAnIdlePoolConnection) — skips re-testing a connection that was proven healthy within the last N seconds, reducing testing overhead under heavy traffic.
- **Count of Test Failures Till Flush** (CountOfTestFailuresTillFlush) — number of consecutive test failures before WebLogic Server closes every connection in the pool, to cut short repeated, wasted testing (default 2).
- **Count of Refresh Failures Till Disable** (CountOfRefreshFailuresTillDisable) — number of consecutive failures to replace a dead connection before the whole connection pool is suspended (default 2).

4.2 Database Connection Testing Semantics

A connection test reserves a connection, runs a small validation query, and returns the connection to the pool. WebLogic Server tracks statistics — test duration, number of connections waiting, number of connections under test — and uses recent history to decide how long to wait before declaring a test failed.

If a test appears to hang abnormally long (10 seconds by default), the pool may declare a DBMS connectivity failure, disable itself, and terminate all connections — whether idle or actively held by an application. In-use connections are flagged for later verification and, if they still have not progressed once Test Frequency Seconds elapses, are forcibly killed. This behavior is intentionally rare: its purpose is to break otherwise indefinite hangs caused by dropped network cables and similar failures, which the OS TCP stack itself might not resolve for as long as ten minutes.

The query used depends on Test Table Name: a bare table name becomes select count(*) from table_name; a value beginning with SQL followed by a statement uses that statement directly. A connection that fails its test is closed, recreated, and the replacement is tested in turn.

4.2.1 Testing at Connection Creation

Every connection created in a data source — at deployment, at server startup, when the pool grows to meet demand, or when replacing a connection that failed a test — is tested immediately using the Test Table Name query, ensuring it is viable before an application ever sees it.

4.2.2 Periodic Connection Testing

When Test Frequency is greater than 0, WebLogic Server periodically tests idle (unreserved) pooled connections using the Test Table Name query, replacing any that fail before returning them to service.

4.2.3 Testing Reserved Connections

When Test Connections On Reserve is enabled, each connection is validated immediately before being handed to a requesting application. This guarantees a healthy connection at the cost of added latency on every reservation; that cost can be reduced by tuning Seconds to Trust an Idle Pool Connection. This option is disabled by default.

4.2.4 Minimizing Connection Test Delay After Database Connectivity Loss

A momentary loss of DBMS connectivity can terminate some or all connections in a pool. With Test Connections on Reserve enabled, an application request triggers a test, discovers the termination, and attempts to replace the connection — a process that can impose a long delay in some failure modes.

To reduce this delay, WebLogic Server treats a run of consecutive test failures as evidence that every connection in the pool is dead, and proactively closes them all. Subsequent requests then create fresh connections directly, without first testing (and waiting on) a doomed connection.

- If Test Frequency is greater than 0, the failure threshold is CountOfTestFailuresTillFlush (default 2).
- If Test Frequency is 0 (periodic testing disabled), the threshold defaults to 25% of Maximum Capacity, but is overridden by CountOfTestFailuresTillFlush if that value is set.

> ***Note:*** *Requires Test Connections On Reserve to be enabled. Once the threshold is reached, unused connections are terminated immediately; active connections are monitored and destroyed if no activity is detected within 60 seconds. Consider raising this value for slow applications with long gaps in JDBC activity, or for networks with chronic single-connection drops.*

4.2.5 Minimizing Connection Request Delays After Loss of DBMS Connectivity

If the DBMS remains unavailable for an extended period, the data source will keep testing and replacing dead connections on every request — each test can take as long as the network timeout, compounding delay for every dead connection in the pool.

Setting CountOfRefreshFailuresTillDisable (default 2, requires Test Connections On Reserve enabled and Initial Capacity greater than 0) causes WebLogic Server to suspend the entire connection pool once that many consecutive replacement failures occur. Any subsequent request against a suspended pool immediately receives a PoolDisabledSQLException instead of waiting on a doomed test.

While suspended, WebLogic Server runs a background refresh process every 5 seconds (not configurable): it checks database health, and once the database is confirmed recovered, creates a new connection and re-enables the data source. The pool can also be re-enabled manually via the Remote Console or WLST.

> ***Note:*** *When a data source belongs to a Multi Data Source, the Multi Data Source — not the member data source — takes over responsibility for disabling and re-enabling members, by default checking every two minutes (configurable via test frequency seconds at the Multi Data Source level, which has different semantics than at the individual data source level).*

4.2.6 Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection

For workloads that reserve, use, and close connections in rapid short cycles, testing overhead on every cycle can be significant. Seconds to Trust an Idle Pool Connection lets WebLogic Server skip re-testing a connection that was proven healthy (by a prior test or a successful use) within the configured window — applicable both to Test Reserved Connections and to periodic testing.

This is a pure performance/robustness trade-off: a higher value reduces test overhead but increases the odds a request receives a connection that failed silently in the interim.

4.3 Database Connection Testing Configuration Recommendations

Choose testing settings to match your application's tolerance for failures versus delay:

- If your application cannot tolerate a dead-connection failure: set Seconds to Trust an Idle Pool Connection to 0 and enable Test Reserved Connections, so every connection is verified immediately before use.
- If your application is more sensitive to reservation delay and can tolerate an occasional failure from a dead connection: use a higher Seconds to Trust an Idle Pool Connection, a lower Test Frequency, and still enable Test Reserved Connections — relying more on background testing than on per-request testing.

> ***Note:*** *Even with ideal settings, a connection can fail in the instant between a successful test and the application's actual use of it. Applications should always be written to handle unexpected exceptions from a dead connection gracefully.*

With Active GridLink (AGL) and Fast Application Notification (FAN) enabled:

- Test Connections on Reserve is less necessary for detecting a downed database instance, since Oracle Notification Service (ONS) pushes that event directly — reducing testing overhead. Oracle still recommends enabling Test Connections on Reserve (since it also catches network and access issues), paired with SecondsToTrustAnIdlePoolConnection and/or TestFrequencySeconds to keep overhead low.
- CountOfTestFailuresTillFlush and CountOfRefreshFailuresTillDisable are ignored in this mode — an entire RAC instance is instead disabled directly in response to a FAN 'instance down' event.

4.4 Default Test Table Name by DBMS

The WebLogic Remote Console automatically sets a DBMS-appropriate default for Test Table Name when a data source is created. The database user used for connections must have access to whatever table or query is configured; otherwise, either grant access or point Test Table Name at an accessible object.

Test Table Name accepts either a bare table name (WebLogic issues select count(*) from that table — Oracle's DUAL is a good, rarely updated example) or a full statement prefixed with SQL, e.g. SQL select 1, which works well on databases like SQL Server that can select a constant without a FROM clause.

DBMS	Default Test Table Name (Query)
DB2	SQL SELECT COUNT(*) FROM SYSIBM.SYSTABLES
Microsoft SQL Server	SQL SELECT 1
MySQL	SQL SELECT 1
Oracle	SQL ISVALID
Sybase	SQL SELECT 1

For Oracle databases — especially RAC environments — the default Test Table Name setting delivers the best overall performance. Oracle continues to support SQL PINGDATABASE and SQL SELECT 1 FROM DUAL as alternatives; the default SQL ISVALID is less exhaustive than SELECT 1 FROM DUAL but meaningfully reduces overhead and improves SOA-style workload performance.

5. Enabling Connection Creation Retries

Creation Retry Frequency sets the number of seconds between attempts to establish the pool's connections against the database. Without it set, the data source fails to deploy if the database is unreachable at creation time. With it set, WebLogic Server keeps retrying, at the specified interval, until it succeeds.

This setting applies only to connections created when the data source is created (server startup, deployment, or an initial-capacity increase) — it does not apply to connections created later to expand the pool or to replace a defunct connection. The default value is 0 seconds, meaning retries are disabled and data source creation fails outright if the database is unavailable.

6. Enabling Connection Requests to Wait for a Connection

Two attributes work together to let connection requests wait for availability instead of failing immediately, without letting too many waiting threads degrade the system.

6.1 Connection Reserve Timeout

When every connection in a fully-expanded pool is busy, a request normally fails with a Connection Unavailable exception. Setting Connection Reserve Timeout (in seconds) makes the request wait instead, up to that many seconds, for a connection to free up; if none appears in time, the request fails with a PoolLimitSQLException.

- -1 — the request times out immediately if no connection is available.
- 0 — the request waits indefinitely.
- Default: 10 seconds.

6.2 Limiting the Number of Waiting Connection Requests

Each waiting request blocks a thread; too many concurrent waits can degrade the whole system. Maximum Waiting for Connection (HighestNumWaiters) caps how many requests may wait concurrently.

- MAX-INT (default) — effectively unbounded waiting.
- 0 — no request may wait at all; once the limit of allowed requests is reached, a SQLException is thrown immediately for new requests.

7. Automatically Recovering Leaked Connections

A leaked connection is one an application reserved but never properly returned to the pool. Setting Inactive Connection Timeout causes WebLogic Server to forcibly reclaim a reserved connection after it has shown no activity for the configured number of seconds. The default value, 0, disables this feature.

> ***Note:*** *The actual timeout can exceed the configured value. The internal maintenance thread runs every 5 seconds; on reaching the configured Inactive Connection Timeout it gives an apparently-inactive connection a 'second chance' before forcibly reclaiming it on the next check, so the effective delay can average up to 50% longer than configured.*

8. Avoiding Server Lockup with the Correct Number of Connections

To prevent requests from failing simply because the pool has no available connections, ensure Maximum Capacity is set high enough to absorb your application's peak concurrent connection demand. Raising Maximum Capacity increases the ceiling on how far the pool can expand under load.

9. Limiting Statement Processing Time with Statement Timeout

Statement Timeout bounds how long a statement may run on a reserved connection. WebLogic Server passes this value to the JDBC driver via java.sql.Statement.setQueryTimeout(); if the driver rejects the call with an exception, WebLogic Server ignores the requested timeout. Some drivers may silently ignore the call or only partially honor it — check your specific driver's documentation.

The default value, -1, means statements never time out.

10. Using Pinned-To-Thread Property to Increase Performance

Pinned To Thread minimizes the cost of reserving a connection and removes thread contention for connections. When enabled, the first connection an execute thread reserves stays bound to that thread even after the application calls connection.close() — instead of returning to the pool, the connection is simply held for that thread's next request. This eliminates lock contention among threads competing to reserve connections.

> ***Note:*** *Pinned To Thread does not work with an IdentityPool. From WebLogic Server 12.1.2 onward, a data source configured with both will fail to deploy.*

10.1 Behavioral Changes When Pinned To Thread is Enabled

- Maximum Capacity is ignored — the pool's connection count instead equals the greater of Initial Capacity or the number of connections currently reserved.
- Shrinking never applies, since connections are effectively always reserved and never returned to the pool.
- Resetting the pool marks reset connections as Test Needed rather than testing them synchronously; each is tested (and recreated if necessary) the next time it is reserved. Requires Test Connections on Reserve and a configured Test Table Name.
- Combining with Identity Based Connection Pooling Enabled=true causes a deployment error.
- With Use Database Credentials=true, connections remain owned by the default JDBC-descriptor user while the Oracle proxy user/password from getConnection(user, password) is applied — matching non-pinned behavior. The same applies with Oracle Proxy=true.
- Connection labeling is unsupported; requesting a connection with label properties throws an exception.
- With a Multi Data Source (MDS), each member data source maintains its own pinned connections as selected by the MDS algorithm — e.g., with Failover, only the primary member's connections are maintained until a failover occurs; with Load-Balancing, all members' connections are maintained.
- With Active GridLink, Affinity and Runtime Load Balancing continue to choose instances as before; up to one connection per instance per thread is retained (equivalent to a per-instance onePinnedConnectionOnly=true). Gravitation-based migration to lightly loaded nodes is not supported.

10.2 Additional Database Resource Costs

With Pinned To Thread enabled, the pool's effective maximum size becomes (number of execute threads that reserve a connection) × (concurrent connections each thread holds) — which can exceed the configured Maximum Capacity. Since connections are never returned, the pool also can never shrink, so plan for the corresponding database-side resources (e.g., open cursors) accordingly.

The driver property onePinnedConnectionOnly limits this cost: when set to true, only the first connection a thread requests is pinned; any additional connections needed by that thread are taken from and returned to the pool normally.

```
Properties="onePinnedConnectionOnly=true;user=examples"
```

Oracle recommends enabling Pinned To Thread where the system can absorb the additional resource requirements, and disabling it if database resource errors appear after enabling it.

11. Using Unwrapped Data Type Objects

By default, WebLogic Server wraps many JDBC objects returned by the driver, which enables debugging output on method calls, connection-utilization tracking for timeouts, and transparent automatic transaction enlistment and security authorization. Disabling wrapping removes this overhead and provides direct access to native driver objects for a measurable performance gain.

Wrapping is particularly relevant for data types that don't implement a standard interface — Oracle's Array, Blob, Clob, NClob, Ref, SQLXML, and Struct classes — since WebLogic Server cannot generate a transparent dynamic proxy for them the way it can for interface-based types.

> ***Note:*** *Oracle recommends avoiding these concrete classes in favor of standard SQL types or the corresponding Oracle interfaces where possible.*

When wrap-types is set to false, the following are left unwrapped: Array, Blob, Clob, NClob, Ref, SQLXML, Struct, ParameterMetaData (no connection testing performed on it), and ResultSetMetaData (no connection testing, result set testing, or JDBC MT profiling performed on it).

11.1 Disabling Wrapping via the WebLogic Remote Console

- In WebLogic Remote Console, click Edit Tree.
- Expand Services, then select Data Sources.
- On the Summary of Data Sources page, click the data source name.
- Select the Configuration: Connection Pool tab.
- Click Advanced to reveal the advanced connection pool options.
- Under Wrap Data Types, deselect the checkbox to disable wrapping.
- Click Save.
- Select Shopping Cart and click Commit Changes to activate the change.

> ***Note:*** *This change does not take effect immediately — it requires the data source to be redeployed or the server to be restarted.*

11.2 Disabling Wrapping via WLST

```
. . .
jdbcSR = create(dsname,"JDBCSystemResource");
theJDBCResource = jdbcSR.getJDBCResource();
poolParams = theJDBCResource.getJDBCConnectionPoolParams();
poolParams.setWrapTypes(false);
. . .
```

12. Tuning Maintenance Timers

Three system properties govern background maintenance timers used by WebLogic JDBC:

- weblogic.jdbc.gravitationShrinkFrequencySeconds — on Active GridLink data sources, connections may need periodic rebalancing when the distribution across RAC instances drifts from the Runtime Load Balancing percentages in FAN advisories; overweight-instance connections are destroyed and replaced. This runs every 30 seconds by default. A value ≤ 0 disables rebalancing entirely.
- weblogic.jdbc.harvestingFrequencySeconds — controls how often (default 30 seconds) WebLogic Server checks whether the pool has fallen to a specified number of available connections and, if so, releases reserved connections that the application has marked harvestable. A value ≤ 0 disables connection harvesting.
- weblogic.jdbc.securityCacheTimeoutSeconds — controls caching of user-credential checks performed on every connection reservation. A value ≤ 0 disables the cache (re-authenticating every time); a value > 0 caches authentication per user for that many seconds (default 10 minutes). If pool access restrictions change dynamically, users are re-authenticated once after the cache is cleared.

13. JDBC Connection Creation Limits

Some databases cap the rate at which new JDBC connections may be created (for example, 100 connections per second). Large WebLogic Server deployments can exceed such limits during server startup, database rolling restarts, or failover events — and even databases without an explicit rate limit can be overwhelmed by a burst of simultaneous connection requests.

Two connection properties help avoid this:

- weblogic.jdbc.maxConcurrentCreateRequests — caps the number of concurrent connection-create operations.
- weblogic.jdbc.concurrentCreateRequestsTimeoutSeconds — how long (60 seconds by default) a connection-create request waits for a permit to proceed.

14. Quick Reference: Key Tuning Attributes

The table below summarizes the primary connection-pool tuning attributes covered in this document, their defaults, and their purpose, for fast lookup during configuration or troubleshooting.

Attribute	MBean Property	Default	Purpose
Statement Cache Type	StatementCacheType	LRU	Algorithm used to decide which statements stay cached
Statement Cache Size	StatementCacheSize	10	Number of prepared/callable statements cached per connection
Test Frequency	TestFrequencySeconds	0 (off)	Seconds between background tests of idle connections
Test Reserved Connections	TestConnectionsOnReserve	false	Test a connection immediately before handing it to an app
Test Table Name	TestTableName	DBMS-specific	Table or SQL statement used to validate a connection
Seconds to Trust an Idle Pool Connection	SecondsToTrustAnIdlePoolConnection	0	Skip re-testing a connection recently proven healthy
Count of Test Failures Till Flush	CountOfTestFailuresTillFlush	2	Consecutive test failures before the whole pool is flushed
Count of Refresh Failures Till Disable	CountOfRefreshFailuresTillDisable	2	Consecutive failures before the pool is suspended
Connection Creation Retry Frequency	-	0 (disabled)	Seconds between retries to build the pool when DB is down
Connection Reserve Timeout	ConnectionReserveTimeoutSeconds	10	Seconds a request waits for a free connection
Maximum Waiting for Connection	HighestNumWaiters	MAX-INT	Cap on concurrent threads allowed to wait for a connection
Inactive Connection Timeout	-	0 (off)	Forcibly reclaim a connection left idle while reserved (leak recovery)
Maximum Capacity	-	-	Ceiling on the number of physical connections in the pool
Statement Timeout	-	-1 (off)	Max seconds a statement may run before being cancelled
Pinned To Thread	-	false	Bind a connection to an execute thread to remove reservation contention
Wrap Data Types	WrapTypes	true	Wrap driver objects for tracking/debugging vs. raw performance

15. Practical Tuning Recommendations

A few consolidated recommendations drawn from the guidance above:

- Size the statement cache from observed behavior, not guesswork — temporarily oversize it, observe pool statistics, then right-size slightly above the largest value seen.
- Enable Test Connections on Reserve for correctness-sensitive applications, and pair it with a modest Seconds to Trust an Idle Pool Connection to control the added latency.
- Set Maximum Capacity generously enough to absorb peak concurrent demand, and use Connection Reserve Timeout plus Maximum Waiting for Connection to bound how requests behave when the pool is briefly exhausted rather than failing them outright.
- Enable Connection Creation Retry Frequency and the initial-capacity Connection Retry properties in environments where the database may not be available at exactly the moment the data source deploys.
- Use Inactive Connection Timeout to guard against connection leaks in applications, keeping in mind the effective delay can run up to 50% past the configured value.
- Reserve Pinned To Thread for latency-critical, high-throughput scenarios where the database can absorb the larger effective connection count it can produce — and use onePinnedConnectionOnly to bound that cost.
- Disable Wrap Data Types only after confirming your application does not rely on WebLogic's automatic transaction enlistment, security authorization, or debugging instrumentation for the affected types.
- On RAC/Active GridLink deployments, prefer FAN-based failure detection over heavy reliance on Test Connections on Reserve, and tune weblogic.jdbc.gravitationShrinkFrequencySeconds if rebalancing behavior needs adjustment.