Dishtha Yantra Database Framework

Requirements Specification

Version 1.0

*Enterprise-Grade Connection Pool Management*

# 1. System Overview

## 1.1 Purpose

The Dishtha Yantra Database Framework provides an enterprise-grade database connection pooling solution inspired by Apache DBCP (Database Connection Pool). It offers robust connection management, automatic resource cleanup, health monitoring, and support for multiple database platforms with comprehensive configuration options.

## 1.2 Key Features

* Multi-database support: PostgreSQL, MySQL, SQLite, Oracle, SQL Server
* Connection validation and health checking with configurable strategies
* Automatic eviction of idle, expired, and invalid connections
* Abandoned connection detection and recovery
* Multiple eviction policies: Idle Time, Lifetime, Soft Min Idle, LRU
* Singleton pool manager for centralized pool management
* Comprehensive statistics and monitoring capabilities
* Thread-safe operations with fine-grained locking
* Connection lifecycle management with state tracking
* Context manager support for automatic resource cleanup

# 2. Core Architecture

## 2.1 Supported Database Types

The framework supports five major database platforms through standardized interfaces:

| **Database** | **Driver** | **Use Cases** |
| --- | --- | --- |
| **PostgreSQL** | psycopg2 | Production applications, data warehousing |
| **MySQL** | pymysql | Web applications, e-commerce platforms |
| **SQLite** | sqlite3 (built-in) | Testing, embedded systems, local storage |
| **Oracle** | cx\_Oracle | Enterprise applications, banking systems |
| **SQL Server** | pyodbc | Microsoft stack integration, enterprise apps |

## 2.2 Connection State Management

Connections maintain explicit state throughout their lifecycle:

| **State** | **Description** |
| --- | --- |
| **IDLE** | Connection is available in the pool, ready to be borrowed |
| **IN\_USE** | Connection is actively being used by a thread |
| **TESTING** | Connection is undergoing validation testing |
| **INVALID** | Connection has failed validation and is marked for removal |
| **CLOSED** | Connection has been closed and removed from the pool |

## 2.3 Connection Wrapper

The ConnectionWrapper class encapsulates database connections with comprehensive metadata tracking:

| **Attribute** | **Purpose** |
| --- | --- |
| **connection\_id** | Unique UUID identifier for tracking |
| **created\_time** | Timestamp when connection was created |
| **last\_used\_time** | Timestamp of last usage for idle detection |
| **last\_tested\_time** | Timestamp of last validation test |
| **use\_count** | Total number of times connection has been borrowed |
| **error\_count** | Number of errors encountered during validation |
| **in\_transaction** | Boolean flag indicating active transaction state |
| **borrowed\_by** | Reference to thread currently using the connection |

## 2.4 Pool Configuration

The PoolConfig class provides comprehensive configuration options organized into functional categories:

### 2.4.1 Basic Pool Settings

* **min\_idle:** Minimum idle connections to maintain (default: 2)
* **max\_idle:** Maximum idle connections allowed (default: 8)
* **max\_total:** Maximum total connections (default: 20)

### 2.4.2 Connection Validation Settings

* **test\_on\_borrow:** Validate connection before lending (default: True)
* **test\_on\_return:** Validate connection when returned (default: False)
* **test\_while\_idle:** Test idle connections periodically (default: True)
* **validation\_query:** Custom query for validation (default: database-specific)
* **validation\_timeout:** Timeout for validation query in seconds (default: 5)

### 2.4.3 Eviction Settings

* **time\_between\_eviction\_runs:** Seconds between eviction runs (default: 30)
* **min\_evictable\_idle\_time:** Min idle time before eviction (default: 300)
* **max\_connection\_lifetime:** Max connection lifetime (default: 3600)
* **num\_tests\_per\_eviction\_run:** Connections to test per run (default: 3)
* **eviction\_policy:** Eviction strategy (default: IDLE\_TIME)

### 2.4.4 Behavior Settings

* **block\_when\_exhausted:** Block when pool exhausted (default: True)
* **max\_wait\_time:** Max wait time for connection in seconds (default: 30)
* **lifo:** Last-in-first-out for idle connections (default: True)
* **fair:** Fair mode with FIFO for waiting threads (default: False)

### 2.4.5 Monitoring Settings

* **abandoned\_remove:** Remove abandoned connections (default: True)
* **abandoned\_timeout:** Timeout for abandoned detection (default: 300)
* **log\_abandoned:** Log abandoned connection details (default: True)

# 3. Functional Requirements

## 3.1 Connection Pool Operations

### 3.1.1 Connection Borrowing

The pool must support safe and efficient connection borrowing with the following requirements:

* Attempt to retrieve idle connection from queue
* Create new connection if pool not at max capacity
* Block and wait if configured and pool exhausted
* Validate connection before returning if test\_on\_borrow enabled
* Track borrowing thread for abandoned connection detection
* Update connection metadata (state, last\_used\_time, use\_count)
* Record borrowing statistics

### 3.1.2 Connection Return

When a connection is returned to the pool:

* Validate connection state if INVALID state
* Check if connection in active transaction and rollback if needed
* Test connection if test\_on\_return enabled
* Update connection state to IDLE
* Clear borrowing thread reference
* Return to idle queue or destroy if exceeding max\_idle
* Record return statistics

### 3.1.3 Connection Validation

Connection validation must ensure reliability:

* Set connection state to TESTING during validation
* Execute validation query with timeout
* Use database-specific validation queries (SELECT 1, SELECT 1 FROM DUAL, etc.)
* Increment error\_count on validation failure
* Update last\_tested\_time timestamp
* Record validation statistics

## 3.2 Eviction Policies

The framework supports multiple eviction policies for flexible connection lifecycle management:

### 3.2.1 Idle Time Eviction

* Calculate idle time as current\_time minus last\_used\_time
* Evict connections idle longer than min\_evictable\_idle\_time
* Maintain minimum idle connections count

### 3.2.2 Lifetime Eviction

* Calculate age as current\_time minus created\_time
* Evict connections exceeding max\_connection\_lifetime
* Prevent connection reuse beyond safe lifetime

### 3.2.3 Soft Min Idle

* Evict idle connections only if idle count exceeds min\_idle
* Combine with idle time checks for gentle eviction

### 3.2.4 LRU (Least Recently Used)

* Sort idle connections by last\_used\_time
* Evict oldest used connections first

## 3.3 Abandoned Connection Detection

The system must detect and recover connections that were borrowed but never returned:

* Track borrowing thread for each active connection
* Check if borrowed connection exceeds abandoned\_timeout
* Verify borrowing thread is still alive
* Log abandoned connection details if log\_abandoned enabled
* Forcibly reclaim abandoned connections
* Return reclaimed connections to idle pool or destroy if needed

## 3.4 Pool Manager Operations

### 3.4.1 Pool Creation and Reuse

The pool manager must efficiently manage multiple pools:

* Generate unique key from connection configuration (SHA256 hash)
* Check if pool exists for given configuration
* Reuse existing pool if configuration matches
* Create new pool only when needed
* Enforce maximum pool limit (default: 50 pools)
* Track pool access statistics and timestamps

### 3.4.2 Named Pool Management

* Support user-defined pool names for easy reference
* Maintain mapping of names to pool keys
* Allow retrieval by name or configuration
* Support pool removal by name

### 3.4.3 Pool Cleanup

* Background thread for periodic cleanup
* Remove pools idle longer than pool\_idle\_timeout
* Check pool reference count before removal
* Gracefully close pools before removal
* Update cleanup statistics

# 4. Monitoring and Statistics

## 4.1 Pool-Level Statistics

Each connection pool maintains comprehensive statistics:

| **Metric** | **Description** |
| --- | --- |
| connections\_created | Total connections created since pool initialization |
| connections\_destroyed | Total connections destroyed |
| connections\_borrowed | Total borrow operations |
| connections\_returned | Total return operations |
| connections\_validated | Total validation operations performed |
| connections\_invalidated | Connections marked invalid and removed |
| average\_wait\_time | Average time threads wait for connections |
| max\_wait\_time | Maximum wait time recorded |

## 4.2 Manager-Level Statistics

The pool manager tracks system-wide metrics:

* **total\_pools\_created:** Cumulative count of all pools created
* **total\_pools\_destroyed:** Cumulative count of all pools destroyed
* **total\_connections\_borrowed:** Aggregated borrow count across all pools
* **total\_connections:** Current total connections across all pools
* **per\_pool\_details:** Individual pool statistics and status
* **named\_pools:** List of user-defined pool names

## 4.3 Health Monitoring

The framework provides comprehensive health checking capabilities:

* Test connection retrieval from each pool
* Execute validation query on borrowed connection
* Classify pools as healthy or unhealthy
* Report degraded status if any pools unhealthy
* Provide detailed error information for unhealthy pools

# 5. API Specifications

## 5.1 DatabaseConnectionPool API

| **Method** | **Parameters** | **Description** |
| --- | --- | --- |
| **borrow\_connection()** | timeout | Borrow connection from pool |
| **return\_connection()** | wrapper | Return connection to pool |
| **get\_connection()** | timeout | Context manager for auto return |
| **validate\_connection()** | wrapper | Validate connection health |
| **invalidate\_connection()** | wrapper | Mark connection invalid |
| **get\_pool\_status()** | None | Get pool status and statistics |
| **clear\_idle\_connections()** | None | Clear all idle connections |
| **close()** | None | Close pool and all connections |

## 5.2 DBConnectionPoolManager API

| **Method** | **Parameters** | **Description** |
| --- | --- | --- |
| **get\_pool()** | connection\_config, pool\_config | Get or create pool for config |
| **get\_pool\_by\_name()** | name, connection\_config | Get or create named pool |
| **get\_connection()** | db\_type, \*\*kwargs | Get connection directly |
| **remove\_pool()** | pool\_key | Remove and close pool |
| **get\_statistics()** | None | Get manager statistics |
| **health\_check()** | None | Check health of all pools |
| **clear\_idle\_pools()** | max\_idle\_time | Remove idle pools |
| **shutdown()** | None | Shutdown manager and all pools |

# 6. Non-Functional Requirements

## 6.1 Performance

* Connection borrowing must complete within milliseconds under normal load
* Queue operations must use thread-safe, non-blocking algorithms
* Validation overhead minimized through intelligent test intervals
* Eviction runs must not block connection operations
* Statistics tracking with minimal overhead using atomic operations

## 6.2 Reliability

* Automatic recovery from database disconnections
* Connection validation before use prevents stale connections
* Abandoned connection detection prevents resource leaks
* Graceful degradation when maximum pool size reached
* Transaction rollback on connection return ensures clean state

## 6.3 Scalability

* Support hundreds of concurrent connections per pool
* Manage up to 50 pools simultaneously through pool manager
* Configurable pool sizes to match application needs
* Efficient memory usage through connection reuse

## 6.4 Thread Safety

* All pool operations thread-safe using appropriate locking
* Queue operations inherently thread-safe
* Statistics tracking uses thread-safe counters
* Manager uses singleton pattern with double-checked locking
* No race conditions in connection lifecycle management

## 6.5 Maintainability

* Clean separation of concerns between pool and manager
* Comprehensive logging at appropriate levels
* Dataclasses for clear configuration structure
* Enum types for state management and policies
* Well-documented public API

## 6.6 Security

* Password not included in pool key generation
* SSL/TLS support for all compatible databases
* Connection credentials managed securely
* Support for certificate-based authentication

## 6.7 Monitoring and Observability

* Real-time pool status available via API
* Comprehensive statistics for performance analysis
* Health check endpoint for external monitoring
* Detailed logging for troubleshooting
* Per-connection metadata tracking

# 7. Usage Patterns

## 7.1 Basic Pool Usage

# Create pool with configuration

config = PoolConfig(min\_idle=5, max\_idle=10, max\_total=20)

pool = DatabaseConnectionPool(

DatabaseType.POSTGRESQL, config,

host='localhost', database='mydb',

user='user', password='pass'

)

# Use with context manager

with pool.get\_connection() as conn:

cursor = conn.cursor()

cursor.execute('SELECT \* FROM users')

results = cursor.fetchall()

## 7.2 Pool Manager Usage

# Get singleton manager

manager = DBConnectionPoolManager()

# Create connection configuration

config = ConnectionConfig(

db\_type=DatabaseType.MYSQL,

host='localhost', database='app\_db',

user='app\_user', password='secret'

)

# Get or create pool (reuses existing)

pool = manager.get\_pool(config)

# Or get connection directly

with manager.get\_connection(

db\_type='postgresql',

host='localhost', database='mydb'

) as conn:

# Use connection

## 7.3 Named Pools

# Create named pool for easy reference

analytics\_pool = manager.get\_pool\_by\_name(

'analytics',

ConnectionConfig(

db\_type=DatabaseType.POSTGRESQL,

host='analytics.db.com',

database='analytics'

)

)

# 8. Implementation Notes

## 8.1 Queue Selection Strategy

The framework uses Python's Queue.Queue for thread-safe connection management. The LIFO (Last-In-First-Out) mode is preferred by default as it keeps recently used connections active, improving cache locality and reducing the need for validation.

## 8.2 Connection Key Generation

Connection configurations are hashed using SHA256 to create unique keys. Passwords are intentionally excluded from the key for security reasons. This allows pool reuse while maintaining security.

## 8.3 Eviction Thread Design

The eviction thread runs as a daemon thread, performing periodic maintenance without blocking main operations. It processes a limited number of connections per run to distribute load and avoid performance spikes.

## 8.4 Abandoned Connection Detection

Abandoned connections are detected by tracking the borrowing thread and checking if it's still alive. The system uses weak references to avoid preventing thread garbage collection.

## 8.5 Database-Specific Considerations

Each database type has specific connection requirements and validation queries. PostgreSQL uses 'SELECT 1', Oracle uses 'SELECT 1 FROM DUAL', and SQLite may not require validation at all due to its in-process nature.

## 8.6 Singleton Pattern Implementation

The pool manager uses double-checked locking for thread-safe singleton initialization. The \_initialized flag prevents multiple initialization attempts even when multiple threads access the singleton simultaneously.