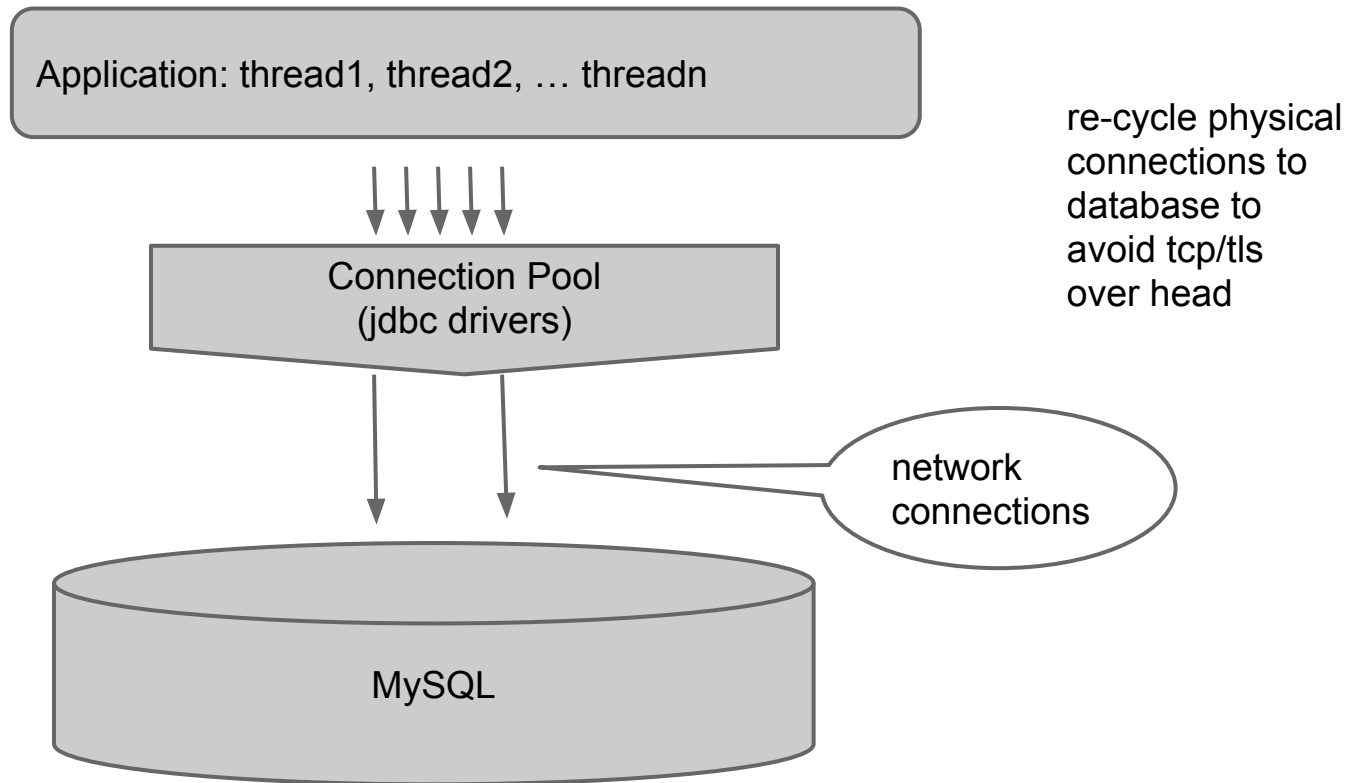


# BoneCP Study

Tech Talk by Jin Peng

# Connection Pool




# Why creating good pool is hard

- Trade of between race conditions and efficiency. Highly concurrent library, naturally **Multi-Threaded**
- Network Resource Leak, keeping track of long lived **Network Connections** is HARD
- Memory Leak, keeping track of long lived **Connection Objects** is HARD

# BoneCP Threads

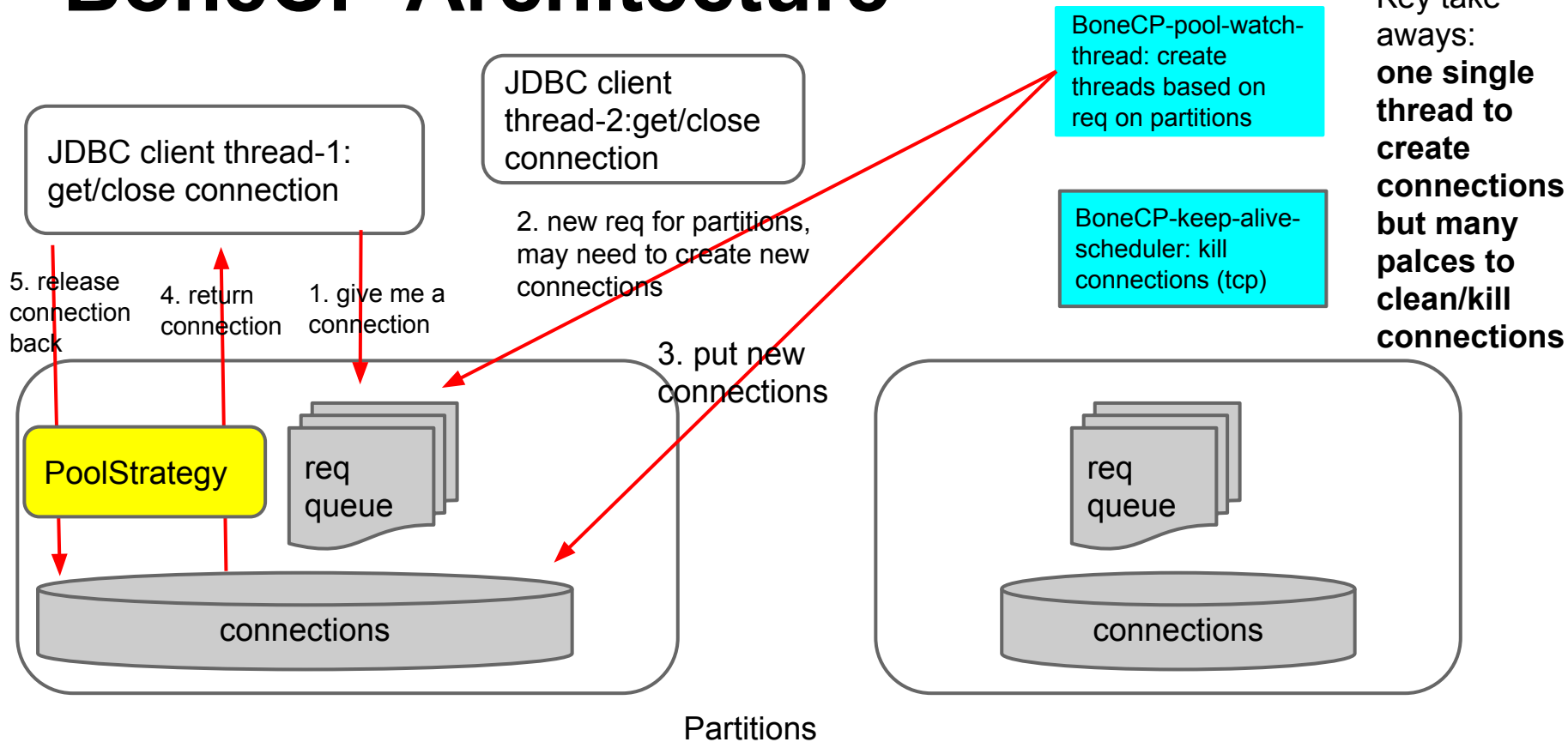
1. BoneCP-keep-alive-scheduler: checks if a connection is still alive, keep dead and send keep alive “select 1” to mysql on schedule
2. BoneCP-pool-watch-thread: create new connection based on demand queue in partitions
3. BoneCP-max-alive-scheduler (optional): prune the connections that have been there for too long
4. BoneCP-connection-watch-thread (optional): monitor a given thread to see it has hold on a connection without close for too long
5. The finalizer thread from google Guava lib to make sure dead connections are released before it gets garbage connected



A screenshot of a Java thread dump showing several threads. The threads are listed with a green triangle icon, a gear icon, and their names in brackets followed by their status in parentheses. The threads are: Daemon Thread [Thread-11] (Running), Thread [Thread-13] (Running), Thread [910299607@qtp-344610950-0] (Running), Thread [1830458131@qtp-344610950-1] (Running), Daemon Thread [Timer-1] (Running), Daemon Thread [Mail Sender Timer] (Running), Daemon Thread [com.google.common.base.internal.Finalizer] (Running), Daemon Thread [BoneCP-keep-alive-scheduler] (Running), and Daemon Thread [BoneCP-pool-watch-thread] (Running). The thread [com.google.common.base.internal.Finalizer] is highlighted with a grey background.

- ▶ ⚙ Daemon Thread [Thread-11] (Running)
- ▶ ⚙ Thread [Thread-13] (Running)
- ▶ ⚙ Thread [910299607@qtp-344610950-0] (Running)
- ▶ ⚙ Thread [1830458131@qtp-344610950-1] (Running)
- ▶ ⚙ Daemon Thread [Timer-1] (Running)
- ▶ ⚙ Daemon Thread [Mail Sender Timer] (Running)
- ▶ ⚙ Daemon Thread [com.google.common.base.internal.Finalizer] (Running)
- ▶ ⚙ Daemon Thread [BoneCP-keep-alive-scheduler] (Running)
- ▶ ⚙ Daemon Thread [BoneCP-pool-watch-thread] (Running)

# BoneCP Architecture



# BonCP classes

- Each BoneCPDataSource create one BoneCP main classe
- BoneCP main class creates monitoring threads, and partitions
- Application getConnection() from BoneCPDataSource
- **Applications that have multiple data sources need multiple instances of all above**

# ConnectionPartition

- Divide the connection pool into Partitions to reduce contention
- Two Bounded

**BlockingQueue<ConnectionHandle>:**

```
--- freeConnections = new LinkedBlockingQueue<ConnectionHandle>(this.  
config.getMaxConnectionsPerPartition())
```

```
--- poolWatchThreadSignalQueue=new ArrayBlockingQueue<Object>(1)
```

# Get Connection Double poll

@Override

protected Connection getConnectionInternal() throws SQLException

ConnectionHandle result = pollConnection();

// we still didn't find an empty one, wait forever (or as per config) until our partition is free

if (result == null) {

int partition = (int) (Thread.currentThread().getId() % this.pool.partitionCount);

ConnectionPartition connectionPartition = this.pool.partitions[partition];

try {

result = connectionPartition.getFreeConnections().poll(this.pool.connectionTimeoutInMs,

TimeUnit.MILLISECONDS);

...

first poll, non-block,  
signal demand  
queue

second poll,  
blocking, wait until  
timeout



# BonCP Killer Thread (BoneCP-keep-alive-scheduler)

```
pcadmin@com-test-den-wwwjdk7-14-9-4-clicktale-e5e500f5-199-209:~$ netstat -an | grep 3306
```

tcp	0	0	172.16.199.209:45239	172.16.162.100:3306	ESTABLISHED
tcp	0	0	172.16.199.209:33581	172.16.162.100:3306	ESTABLISHED
tcp	0	0	172.16.199.209:33578	172.16.162.100:3306	ESTABLISHED
tcp	0	0	172.16.199.209:33576	172.16.162.100:3306	ESTABLISHED
tcp	0	0	172.16.199.209:45241	172.16.162.100:3306	ESTABLISHED
tcp	0	0	172.16.199.209:45145	172.16.162.100:3306	TIME_WAIT
tcp	0	0	172.16.199.209:45149	172.16.162.100:3306	TIME_WAIT
tcp	0	0	172.16.199.209:33580	172.16.162.100:3306	ESTABLISHED
tcp	0	0	172.16.199.209:33561	172.16.162.100:3306	ESTABLISHED
tcp	0	0	172.16.199.209:45244	172.16.162.100:3306	ESTABLISHED
tcp	0	0	172.16.199.209:33577	172.16.162.100:3306	ESTABLISHED
tcp	0	0	172.16.199.209:45146	172.16.162.100:3306	TIME_WAIT

# Pool Strategy

DefaultPoolStrategy: FIFO

CachePoolStrategy: Thread Local First

# Busy Waiting Race Condition in BoneCP-pool-

watch-thread

```
// loop for spurious interrupt
    while (maxNewConnections == 0 || (this.partition.getAvailableConnections()
*100/this.partition.getMaxConnections() > this.poolAvailabilityThreshold)){
        if (maxNewConnections == 0){
            this.partition.setUnableToCreateMoreTransactions(true);
        }

        this.partition.getPoolWatchThreadSignalQueue().take();
        maxNewConnections = this.partition.getMaxConnections()-this.partition.
getCreatedConnections();

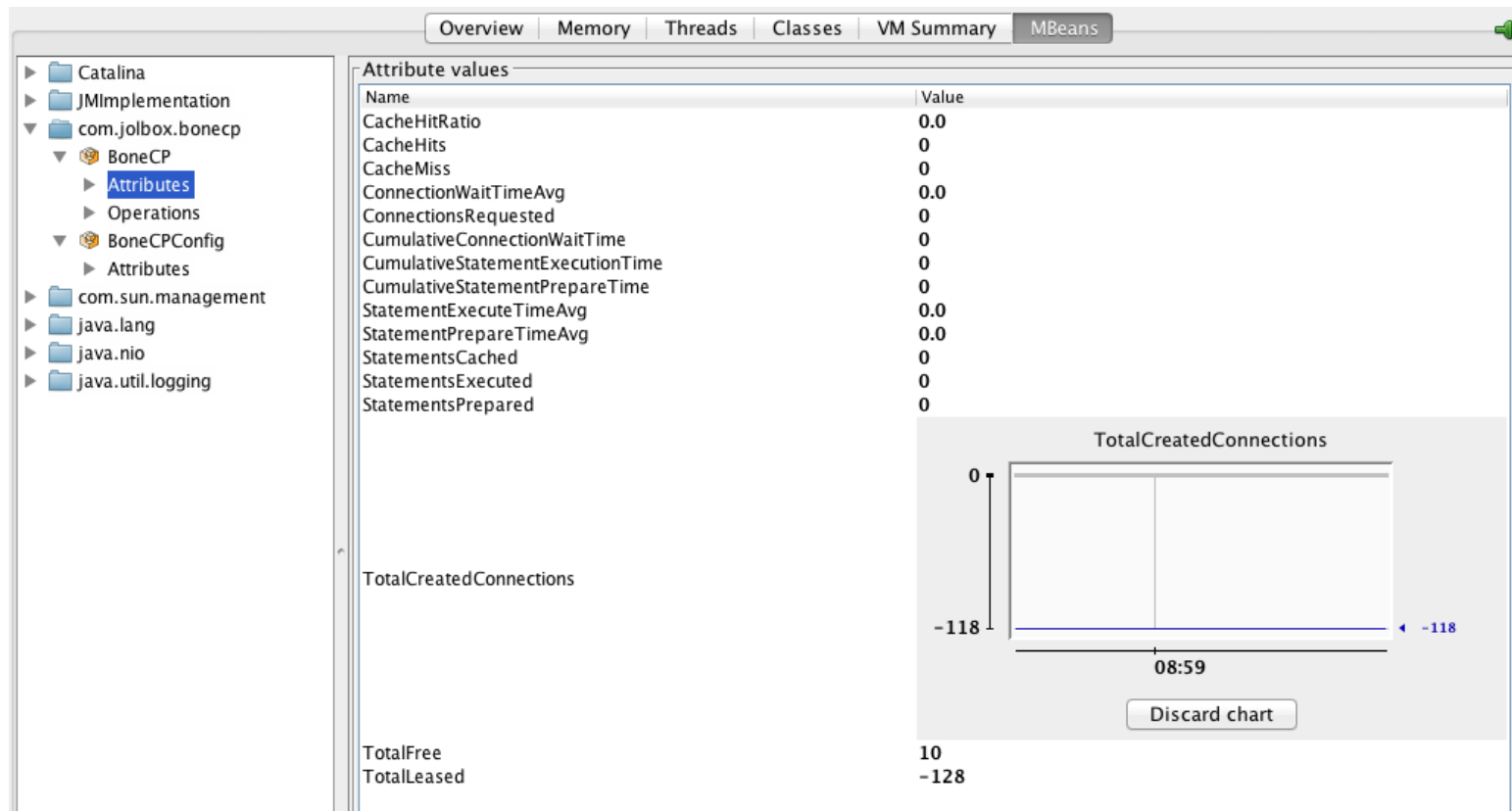
    }
```

# Race on createdConnections counter

Members calling 'updateCreatedConnections(int)' – in workspace

- ▼ updateCreatedConnections(int) : void – com.jolbox.bonecp.ConnectionPartition
  - ▼ addFreeConnection(ConnectionHandle) : void – com.jolbox.bonecp.ConnectionPartition (2 matches)
    - ▶ BoneCP(BoneCPConfig) – com.jolbox.bonecp.BoneCP
    - ▶ fillConnections(int) : void – com.jolbox.bonecp.PoolWatchThread
    - ▶ testFreeConnection() : void – com.jolbox.bonecp.TestConnectionPartition
    - ▶ testFreeConnectionFailing() : void – com.jolbox.bonecp.TestConnectionPartition
    - ▶ testRunCreateConnections() : void – com.jolbox.bonecp.TestPoolWatchThread
  - ▼ finalizeReferent() : void – com.jolbox.bonecp.ConnectionPartition.trackConnectionFinalizer(...).new FinalizableWeakReference() {...}
    - ▼ [constructor] new FinalizableWeakReference() {...} – com.jolbox.bonecp.ConnectionPartition.trackConnectionFinalizer(...)
      - ▶ trackConnectionFinalizer(ConnectionHandle) : void – com.jolbox.bonecp.ConnectionPartition
- ▼ [callers]
  - ▶ cleanUp() : void – com.google.common.base.FinalizableReferenceQueue
- ▼ postDestroyConnection(ConnectionHandle) : void – com.jolbox.bonecp.BoneCP
  - ▶ closeConnection(ConnectionHandle) : void – com.jolbox.bonecp.ConnectionMaxAgeThread
  - ▶ closeConnection(ConnectionHandle) : void – com.jolbox.bonecp.ConnectionTesterThread
  - ▶ destroyConnection(ConnectionHandle) : void – com.jolbox.bonecp.BoneCP
  - ▶ internalReleaseConnection(ConnectionHandle) : void – com.jolbox.bonecp.BoneCP
  - ▶ testCloseConnection() : void – com.jolbox.bonecp.TestConnectionThreadTester
  - ▶ testCloseConnectionNormalCase() : void – com.jolbox.bonecp.TestConnectionMaxAgeTester
  - ▶ testCloseConnectionWithException() : void – com.jolbox.bonecp.TestConnectionMaxAgeTester
  - ▶ testCloseConnectionWithException() : void – com.jolbox.bonecp.TestConnectionThreadTester
  - ▶ testCloseConnectionWithExceptionCoverage() : void – com.jolbox.bonecp.TestConnectionMaxAgeTester
  - ▶ testCloseConnectionWithExceptionInLogger() : void – com.jolbox.bonecp.TestConnectionThreadTester
  - ▶ testConnectionExpired() : void – com.jolbox.bonecp.TestConnectionMaxAgeTester
  - ▶ testConnectionMarkedBroken() : void – com.jolbox.bonecp.TestConnectionThreadTester
  - ▶ testIdleConnectionFailedKeepAlive() : void – com.jolbox.bonecp.TestConnectionThreadTester
  - ▶ testIdleConnectionIsKilled() : void – com.jolbox.bonecp.TestConnectionThreadTester
  - ▶ testIdleConnectionIsKilledWithFailure() : void – com.jolbox.bonecp.TestConnectionThreadTester

# Negative TotalCreateConnections



# **Race condition on close connections**

too many places to handle close connections,  
as a result, the numConnectionCreated could  
go negative

# Idle Connection timeout not working

Connections that have been idle for too long are killed, the problem is new connections are added back immediately.

So if the pool size has ever reach full, it will never come back down

# Pool name and JMX

In the BoneCP design, each data source has its own pool and the threads to manage the pool health.

Multiple data sources need to specify unique pool names to show the stat in JMX



# FinalizableReferenceQueue and finalizer thread

```
connectionHandle.getPool().getFinalizableRefs().put(internalDBConnection, new FinalizableWeakReference<ConnectionHandle>(connectionHandle, connectionHandle.getPool().getFinalizableRefQueue())) {
```

```
    @SuppressWarnings("synthetic-access")
```

```
    public void finalizeReferent() {
```

```
        try {
```

```
            pool.getFinalizableRefs().remove(internalDBConnection);
```

```
            if (internalDBConnection != null && !internalDBConnection.isClosed()){// safety!
```

```
                logger.warn("BoneCP detected an unclosed connection "+ConnectionPartition.this.poolName + "and will  
now attempt to close it for you. " +
```

```
                "You should be closing this connection in your application - enable connectionWatch for additional  
debugging assistance or set disableConnectionTracking to true to disable this feature entirely.");
```

```
                internalDBConnection.close();
```

```
                updateCreatedConnections(-1);
```

```
            }
```

```
        } catch (Throwable t) {
```

```
            logger.error("Error while closing off internal db connection", t);
```

```
        }
```

```
    }
```

```
});
```

Other Useful Stuff

# MemorizeTransactionProxy

```
/** This code takes care of recording and playing back of transactions (when a failure occurs). The idea behind this is to wrap a connection
```

```
 * or statement with proxies and log all method calls. When a failure occurs, thrash the inner connection, obtain a new one and play back
```

```
 * the previously recorded methods.
```

```
 *
```

```
 * @author wwadge
```

```
 *
```

```
 */
```

```
public class MemorizeTransactionProxy implements InvocationHandler {
```

## Proposed Fixes

# Long Term Fixes

1. Switch to more robust open source CP, which one?
2. Maybe it is time to write our own open sourced pool project?
  - a. Handle multiple DataSource efficiently
  - b. Handle ReadOnly/ReadWrite DataSource routing natively
  - c. Handle load balancing natively
  - d. Handel network failure/recover gracefully
  - e. Handle Transaction Re-try natively
  - f. Single entry point to create/deploy connections to avoid resource/memory leak
  - g. Easy tracing/monitoring
  - h. Dynamic data source routing based on tenant/regional sharding