Connection Pool Sizing Concepts
The Impact of Non-DB Time

Toon Koppelaars
Real-World Performance
Oracle Server Technologies
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
Who Am I?

- Part of Oracle eco-system since 1987
- #SmartDB proponent
- Coauthor “Applied Mathematics for Database Professionals”
- Now in Real-World Performance group
#1 Issue in Real-World: Database Over-Subscription

• You might be having this problem without knowing it

• Majority of customers that come to us with escalations, experience this

• It’s not obvious that symptoms point to this problem

• Symptoms might lead you down wrong path and spend lot of time there

Opening SR’s with Oracle support, without getting anywhere
Majority DBA?

• Dilemma:
  – You can’t fix this issue by tweaking at database-level
  – Developers need to change how they build applications

• Game plan:
  – Explain underlying application architecture concepts
  – Explain what developers are doing wrong
  – You convey that message back home to your developers

• Developer using connection pools? → listen carefully
Topics

• Preliminaries
  – DB Time and Getting Database Busy
  – CPU Over-subscription

• Web Architecture
  – Connection Pool and Application Threads
  – Non-DB Time in Pooled Database Session

• Over-Subscription, How it Manifests Itself

• Appropriate Connection Pool Size (Some Math)
Preliminary: DB Time
Central Concept: DB Time

• Time spent in database calls by foreground sessions
  
  – Includes CPU time, IO time and wait event times during database call
  
  – Excludes idle time outside database calls
Active Foreground Session

Notice this session is idle approx. > 50%

Wallclock Time

Sum(green) = DB Time

= time spent in database

In application code

In database call
Getting Database Server Busy: The Impact of Non-DB Time

• Let’s assume all DB Time is DB CPU

• How many sessions does it take to get a 10-core DB-server busy?
  – Ten active sessions right?

• Suppose sessions are idle on average for 50% of time
  – Then twenty sessions right? Two per core.

• The higher %-idle in database session, the more sessions you’ll need to get the database busy
Preliminary: CPU Over-Subscription

- The phase prior to database over-subscription
Depending on idle-time in sessions, you’ll need more than 10 sessions to have 10 active sessions.
Topics

• Preliminaries
  – DB Time and Getting Database Busy
  – CPU Over-subscription

• Web Architecture
  – Connection Pool and Application Threads
  – Non-DB Time in Database Session

• Over-Subscription, How it Manifests Itself

• Appropriate Connection Pool Size (Some Math)
Web: Past 15 Years

Pre-loaded application code (ear/jar/war files)

Connection pool

Multi-threaded JVM (thread pool)

Number of pre-spawned connections into database

Using dedicated servers (what we see most)

Dedicated Server

Dedicated Server

Dedicated Server

Database Instance

H T T P - S R V

C1

C2

C3

Multi-threaded JVM (thread pool)
Active Browsers ➔ Application Threads

Eight browsers currently in a request on application server
Introducing Term: “Acquire-Release Cycle”

• Also know as “connection reservation” by application thread
Connection Pool

Executes business logic and needs to call database

T1 T2 T3 T4 T5 T6 T7 T8

Connection pool

C1 C2 C3

JDBC

JVM

S1 S2 S3
Connection Pool: Acquire-Release Cycle

Thread 6 requests and acquires free connection from pool

C2 now unavailable for other threads
Connection Pool: Acquire-Release Cycle

Thread 6 submits SQL via C2

DB session S2 wakes up from idle state
Connection Pool: Acquire-Release Cycle

S2 executes SQL statement
Connection Pool: Acquire-Release Cycle

S2 sends result back to C2

S2 idle again
Connection Pool: Acquire-Release Cycle

C2 sends result back to T6
Connection Pool: Acquire-Release Cycle

T6 releases C2 back to pool

C2 available again for other threads

Connections in pool are time-shared among all browsers

Connection pool

C1 C2 C3

JVM

S1 S2 S3

ORACLE
Points to Make

• **Acquiring free connection by thread = cheap operation**
  – Finding free connection by pool manager → inspecting + updating linked list in Java
  – Handing over connection object reference to application thread
  – Application thread then uses it

• **Giving back used connection to pool = relatively cheap operation**
  – Removing database-state: mainly closing cursors
  – Signaling pool manager "connection is available again"
  – Application thread lets go of reference
Connection Pool: At Full Capacity

Assuming connection pool has been configured to not grow beyond three connections.

At most three threads can be active in database simultaneously.

What if a 4th thread wants to do DB work?
Points to Make

• What if no connection inside pool is available?
  – Application thread requests connection object
  – Pool manager will queue thread’s request (configurable timeout)
    • Note: you must configure this!
  – Puts thread into (Java) wait → called: “connection queuing”
  – Notifies/wakes-up thread once connection becomes available

• Waiting for a connection to become available = also cheap operation

• In general, connection queuing on application server is considered to be undesirable
  – But the alternative, incurring waits on database server, is worse...
Connection Pool Configuration

WebLogic: Console->Services->Data Sources

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Capacity</td>
<td>10</td>
<td>The number of physical connections to create when creating the connection pool in the data source. If unable to create this number of connections, creation of the data source will fail. <a href="#">More Info...</a></td>
</tr>
<tr>
<td>Maximum Capacity</td>
<td>150</td>
<td>The maximum number of physical connections that this connection pool can contain. <a href="#">More Info...</a></td>
</tr>
<tr>
<td>Minimum Capacity</td>
<td>10</td>
<td>The minimum number of physical connections that this connection pool can contain after it is initialized. <a href="#">More Info...</a></td>
</tr>
<tr>
<td>Maximum Waiting for Connection</td>
<td>2147483647</td>
<td>The maximum number of connection requests that can concurrently block threads while waiting to reserve a connection from the data source's connection pool. <a href="#">More Info...</a></td>
</tr>
<tr>
<td>Connection Reserve Timeout</td>
<td>10</td>
<td>The number of seconds after which a call to reserve a connection from the connection pool will timeout. <a href="#">More Info...</a></td>
</tr>
</tbody>
</table>
The Big Question

- What is the appropriate size for my connection pool?

- (Too) Small connection pool size will induce connection queuing in mid-tier
  - You might not be able to use all database server capacity

- (Too) Large connection pool size will induce over-subscription in DB-tier
  - You might over-subscribe the CPU
  - Or worse over-subscribe the database
  - And then you might get outages...
Web Application Architecture

• We need to go into more detail:
  
  Web applications past 15 years → Model-View-Control (MVC)
Model-View-Control (MVC)

• Application code is subdivided into three categories
  – Model: business logic dealing with data
    • Querying data
    • Manipulating data
  – View: application logic dealing with UI
    • Painting UI
    • Responding to UI events
  – Control: page-flow / glue between Model and View
    • Given incoming request, what Model-code should be executed?
    • Given outcome of Model-code execution, what UI changes should be sent back?

• Developers use frameworks for this (M-, V- and C- frameworks)
Model View Control (MVC)

- Incoming request
- C determines what to do
- Calls M module to execute TX

- C determines next page in flow
- Calls M module to retrieve data for it

- C calls V module to render actual html
- C forwards html to client

JVM

C1 C2 C3

S1 S2 S3
Why All This Fuss?

• Because I want to talk about *non-DB time* in pooled database session

• Non-DB time:
  – Time not spent in database
  – Between two subsequent database-requests from *pooled connection object*
  – Mainly driven by *application processing time*
Our Quest: Appropriate Size of Connection Pool

• Appropriate size = just before database CPU becomes over-subscribed

  – We’ll use connection pool max-size to cap/throttle load from APP- to DB-server

  – Assumption: plenty of application threads wanting to do database-work
    • Busy system

  – This implies:
    • Connection pool is constantly fully in use
    • Application threads are acquiring/releasing all the time, maybe waiting bit in connection pool queue
Acquire / Release Connection: per M-module

Black line represents non-DB time
- MVC-code execution
- Time not spent in DBMS
- Between subsequent DB requests
Acquire / Release Connection: per M-module

Even within reservation, there is always non-DB time between DB-calls

And some here

And here

And here

And here

And here
Point to Be Made

• Given a web-request being processed by Java thread, then from database point of view there is:

  – Non-DB time outside pooled connection usage by Java application thread

  – Non-DB time during pooled connection usage by Java application thread
    • We’ll call this: “Non-DB time inside connection reservation”
    • When all connections constantly in use: this is the only driver for idle time in database sessions
Acquire / Release Connection: Per SQL Statement (non-TX)
Acquire / Release Connection: Done by Controller
Acquire / Release Connection

Many strategies for acquire/release connection
Each resulting in different non-DB time inside reservation

Which approach do you think is best?
Do you know which approach is used by your application?
Do your developers know?
There’s One Other Factor Involved: The Network

Time spent on network can be a significant contributor to non-DB Time during reservation!
Some Simple Math

• How can we determine an appropriate connection pool size?
  Given:
  – Number of cores available to database instance
  – No shortage of Java threads wanting to acquire connection
  – Known % non-DB time inside reservation

• We've seen:
  Non-DB time inside reservation of 50% → 2 connections per core required
Connection Pool Sizing

- 0% non-DB time inside reservation → 1 connection per core required
- 80% non-DB time inside reservation → 5 connections per core required
- 90% non-DB time inside reservation → 10 connections per core required
- 95% non-DB time inside reservation → 20 connections per core required

- Appropriate connection pool size := \( \frac{100}{100-X} \) * #cores

X = % non-DB Time inside reservation

Going beyond this size is entering database over-subscription land
Database Over-Subscription

- Too big connection pool causes bad things
  - More overhead in OS
    - More OS processes to schedule
  - More overhead in DBMS
    - Mutexes/Latches/Locks held longer
      - Due to premature de-scheduling from CPU of dedicated server process by OS
    - Chances of wait-events happening, rapidly increase
Over-Subscription / Likely Scenario

DB Time/sec

#cores in DB server

Contention in DB

Useful work done

Run-queue waits

#cores

# active sessions
Over-Subscription / Likely Scenario

DB Time/sec

Run-queue waits

Contention in DB

Useful work done

#cores in DB server

#cores

# active sessions

Copyright © 2017, Oracle and/or its affiliates. All rights reserved. | 45
Over-Subscription / Likely Scenario

- **DB Time/sec**

- **# cores in DB server**

- **# cores**

- **# active sessions**

**Status quo here**
Observation: CPU available! And lots of DB-waits

- **Run-queue waits**
- **Contention in DB**
- **Useful work done**

**Available CPU will invite you to further increase #threads and connection pool**

**Ever increasing waits in DBMS**
Adding sessions won't increase DB-CPU

**CPU overhead in DBMS**

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.
Only One Thing You Should Do

- Decrease and cap #connections to get more useful work done

![Graph showing DB Time/sec vs. #active sessions, with labels for Run-queue waits, Contention in DB, and Useful work done.]
Waiting in Mid-Tier Versus Waiting in DB-Tier

• Over-subscription in database is very expensive

• Far better for Java threads to queue for available connection in mid-tier
  – Over-subscription in mid-tier (connection queuing) is cheap
  – Java threads will simply wait idly in middle tier

• The formula is good starting point for appropriate connection pool size
  – Needs further adjustment in real-world given queuing theory + not all DB Time is CPU
Let’s Plot This Curve

• #Max. connection pool size := \( \frac{100}{100 - X} \) * #cores
Basic Formula Upper Bound Connection Pool Size

10 Core Database Server

\[ Y = \left( \frac{100}{100-X} \right) \times 10 \]

Sizing connection pool above curve just introduces more overhead.

Sizing connection pool under curve introduces server idle time.
Our Rule-of-Thumb: 2-10 Times Number of Cores

\[ Y = \frac{100}{(100-X)} \times 10 \]

We assume your average %Non-DB-time inside reservation is anywhere between 50% and 90%
Connection Pool Sizes in Real-World

What we often see

Makes only sense when your % Non-DB-Time inside reservation is 99.9%
Why Shrinking Connection Pool Won't Always Work

• If you have very high % non-DB time inside connection reservation
  And you are not aware of that

• Shrinking might disable full use of available CPU power on DB-server
  – And so, won't give expected result
  – Your only option then is to decrease the non-DB time inside reservation
    • I.e.: change the application...
That’s All Very Interesting and All, But …

- What the heck is the “%Non-DB-Time inside reservation” for my app?
A Challenge For You

• Become aware of non-DB time during reservation in your application

• Ideally you’d want this to be instrumented by Java developers in their code

• To create awareness with them too

• However, can you as DBA get a clue?
Finding %Non-DB-Time Inside Reservation

• Assuming you’re still in the safe zone
  – Use test-system, or “quiet” prod-system

• Just do a SQL-trace of one of connection-pool sessions for a minute during representative workload and investigate trace-file

• Likely you’ll be able to spot “acquire” and “release” events via some repetitive pattern. For example:
  – “Release” typically would be XCTEND (commit or rollback)
  – “Acquire” typically starts with “timestamp” during quiet period
    Or immediately after XCTEND
    Or you can spot it via some initialization statement
Example Trace File

*** 2017-02-16 11:22:51.009
PARSE #140293387629512:c=0,e=17,p=0,cr=0, cu=0, mis=0, r=0, dep=0, og=1, plh=1534564159, tim=18431411073127
EXEC #140293387629512:c=999,e=824,p=1, cr=4, cu=2, mis=0, r=0, dep=0, og=1, plh=1534564159, tim=18431411074018
FETCH #140293387629512:c=0,e=11,p=0,cr=0, cu=0, mis=0, r=1, dep=0, og=1, plh=1534564159, tim=18431411074107
FETCH #140293387629512:c=0,e=1,p=0,cr=0, cu=0, mis=0, r=0, dep=0, og=0, plh=1534564159, tim=18431411074389
CLOSE #140293387629512:c=0,e=6,dep=0, type=3, tim=18431411074697
PARSE #140293387610760:c=0,e=13,p=0,cr=0, cu=0, mis=0, r=0, dep=0, og=1, plh=0, tim=18431411074733
EXEC #140293387610760:c=0,e=97,p=0,cr=1, cu=6, mis=0, r=1, dep=0, og=1, plh=0, tim=18431411074847
CLOSE #140293387610760:c=0,e=3,dep=0, type=3, tim=18431411075068
PARSE #140293387630952:c=0,e=12,p=0,cr=0, cu=0, mis=0, r=0, dep=0, og=1, plh=1421812382, tim=18431411075102
EXEC #140293387630952:c=0,e=66,p=0,cr=4, cu=1, mis=0, r=1, dep=0, og=1, plh=1421812382, tim=18431411075185
CLOSE #140293387630952:c=0,e=3,dep=0, type=3, tim=18431411075339
XCTEND: only=0, tim=18431411075364

*** 2017-02-16 11:22:59.999
PARSE #140293387629512:c=0,e=33,p=0,cr=0, cu=0, mis=0, r=0, dep=0, og=1, plh=1534564159, tim=18431420062989
EXEC #140293387629512:c=0,e=888,b=1, cr=4, cu=3, mis=0, r=0, dep=0, og=1, plh=1534564159, tim=18431420063979
FETCH #140293387629512:c=0,e=13,p=0,cr=0, cu=0, mis=0, r=1, dep=0, og=1, plh=1534564159, tim=18431420064082
FETCH #140293387629512:c=0,e=1,p=0,cr=0, cu=0, mis=0, r=0, dep=0, og=0, plh=1534564159, tim=18431420064445
CLOSE #140293387629512:c=0,e=10,dep=0, type=3, tim=18431420064780
PARSE #140293387610760:c=0,e=14,p=0,cr=0, cu=0, mis=0, r=0, dep=0, og=1, plh=0, tim=18431420064818
EXEC #140293387610760:c=0,e=188,p=0,cr=1, cu=6, mis=0, r=1, dep=0, og=1, plh=0, tim=18431420065023
CLOSE #140293387610760:c=0,e=4,dep=0, type=3, tim=18431420065306
PARSE #140293387630952:c=0,e=13,p=0,cr=0, cu=0, mis=0, r=0, dep=0, og=1, plh=1421812382, tim=18431420065340
EXEC #140293387630952:c=0,e=97,p=0,cr=4, cu=1, mis=0, r=1, dep=0, og=1, plh=1421812382, tim=18431420065453
CLOSE #140293387630952:c=0,e=3,dep=0, type=3, tim=18431420065636
XCTEND: only=0, tim=18431420065669
Approximating %Non-DB-Time Inside Reservation

• Investigate one acquire-release block in trace-file
  – Determine DB-time by summing all ”e=“ values (dep=0 only)
  – Determine Elapsed time from difference between first and last “tim=“ values
    • Add e-value from first tim value, as tim values represent “time when completed”

• %Non-DB-time-inside-reservation is: \((\text{Elapsed} - \text{DB-Time})/\text{Elapsed}) \times 100\)
Approximating Think-Time-Inside-Reservation

**2017-02-16 11:22:51.009**

```
*** PARSE #140293387629512: c=0, e=17, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, ph=1534564159, tim=18431411073127
EXEC #140293387629512: c=999, e=824, p=1, cr=4, cu=2, mis=0, r=0, dep=0, og=1, ph=1534564159, tim=18431411074018
FETCH #140293387629512: c=0, e=11, p=0, cr=0, cu=0, mis=0, r=1, dep=0, og=1, ph=1534564159, tim=18431411074107
FETCH #140293387629512: c=0, e=1, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, ph=1534564159, tim=18431411074389
CLOSE #140293387629512: c=0, e=6, dep=0, type=3, tim=18431411074697
PARSE #140293387610760: c=0, e=12, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, ph=18431411074733
EXEC #140293387610760: c=0, e=97, p=0, cr=1, cu=6, mis=0, r=1, dep=0, og=1, ph=18431411074847
CLOSE #140293387610760: c=0, e=3, dep=0, type=3, tim=18431411075068
PARSE #140293387629512: c=0, e=12, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, ph=18431411075102
EXEC #140293387629512: c=0, e=66, p=0, cr=4, cu=1, mis=0, r=1, dep=0, og=1, ph=18431411075185
CLOSE #140293387629512: c=0, e=3, dep=0, type=3, tim=18431411075339
XCTEND rlbk=0, rd_only=0, tim=18431411075364
```

**2017-02-16 11:22:59.999**

```
*** PARSE #140293387629512: c=0, e=33, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, ph=1534564159, tim=18431420062989
EXEC #140293387629512: c=0, e=888, p=1, cr=4, cu=3, mis=0, r=0, dep=0, og=1, ph=1534564159, tim=18431420063979
FETCH #140293387629512: c=0, e=13, p=0, cr=0, cu=0, mis=0, r=1, dep=0, og=1, ph=1534564159, tim=18431420064082
FETCH #140293387629512: c=0, e=1, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, ph=18431420064445
CLOSE #140293387629512: c=0, e=10, dep=0, type=3, tim=18431420064518
PARSE #140293387610760: c=0, e=14, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, ph=18431420064718
EXEC #140293387610760: c=0, e=188, p=0, cr=0, cu=0, mis=0, r=1, dep=0, og=1, ph=1843142006523
CLOSE #140293387610760: c=0, e=4, dep=0, type=3, tim=18431420065699
PARSE #140293387630952: c=0, e=13, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, ph=1421812382, tim=18431420065340
EXEC #140293387630952: c=0, e=97, p=0, cr=4, cu=1, mis=0, r=0, dep=0, og=1, ph=1421812302, tim=18431420065453
CLOSE #140293387630952: c=0, e=3, dep=0, type=3, tim=18431420065636
XCTEND rlbk=0, rd_only=0, tim=18431420065669
```

**Difference is:**

- **2,237us**
- **17us**

**= 2,254us**

**Sum is:**

- **1,053us**

$$((2,254 - 1,053)/2,254)*100\% = 53\%$$ approximate non-DB-time inside reservation

⇒ Connection pool size about 2X number of cores
What If You Can’t Approximate, You’re Back to:

- Rule of thumb: size of connection pool is 2-10 times number of CPU cores available

\[ Y = \frac{100}{(100 - X)} \times 10 \]

The complexity of your layered software architecture drives more towards 2X or more towards 10X.
Wrapping Up

• Preliminaries
  – DB Time and Getting Database Busy
  – CPU Over-subscription

• Web Architecture
  – Connection Pool and Application Threads
  – Non-DB Time in Pooled Database Session

• Over-Subscription, How it Manifests Itself

• Appropriate Connection Pool Size (Some Math)
Remember Our Game Plan: Bring This Message Home

- As DBA you can maybe decrease network-time component of non-DB-time
  - Make sure APP + DB servers in same rack and directly connected
- On your next application development effort try to be aware, or better in control, of Acquire/Release cycles, and (MVC) code execution during these cycles
- Your solution should minimize non-DB-time inside reservation
Questions?
Integrated Cloud
Applications & Platform Services