ORACLE® DATABASE 12c

Oracle JDBC Memory Management

Introduction

Database applications can use large amounts of memory. Large scale JDBC applications can run into performance problems due to the amount of memory they use. The Oracle database 10i and 11g JDBC drivers intentionally trade off large memory use for improved performance. The Oracle database 12c drivers are more frugal with memory but large applications can still run into memory problems. This white paper gives some insight into how the various drivers use memory and how to tune them for best performance.

In Oracle database 12c the driver’s memory management has been designed to minimize memory use, maximize performance and support the 32K limit on VARCHAR columns introduced in Oracle database 12c. The 12c memory management scheme imposes some overhead, but the reduced memory footprint typically provides performance equal or better than 10i or 11g. Even so big data applications can use a lot of memory even with the 12c drivers. If your application performs acceptably then there is no reason to worry about memory use. If your application does not have the performance you wish and uses more memory than desirable, read on.

Where Does It All Go?

JDBC drivers use memory for many things, but the big issue is the buffers used to store query results. Each Statement (including PreparedStatement and CallableStatement) has to store the query results in memory. In 12c each Statement holds a set of byte[] buffers to store the query results. In 10i and 11g each Statement holds two buffers, one byte[] and one char[]. The char[] stores all the row data of character type: CHAR, VARCHAR2, NCHAR, etc. The byte[] stores all the rest.

In 10i and 11g the buffers are allocated when the SQL string is parsed, generally the first time the Statement is executed. The Statement holds these two buffers until it is closed. Since the buffers are allocated when the SQL is parsed, the size of the buffers depends not on the actual size of the row data returned by the query, but on the maximum size possible for the row data. After the SQL is parsed, the type of every column is known and from that information the driver can compute the maximum amount of memory required to store each column. The driver also has the fetchSize, the number of rows to retrieve on each fetch. With the size of each column and the number of rows, the driver can compute the absolute maximum size of the data returned in a single fetch. That is the size of the buffers.

In 12c the buffers are allocated on demand as the driver reads the result data from the server. This minimizes the amount of memory required to store the query results at the cost of some additional overhead compared to 10i and 11g.

Some types such as LONG and LONG RAW can be too large to store in line in the buffers and are handled differently. By default, if the query result contains a LONG or LONG RAW, the fetch size is set to 1, which addresses most memory problems as will become clear further on.

In 10i and 11g character data is stored in the char[] buffer. Java chars take up two bytes per character. A VARCHAR2(10) column will contain a maximum of ten characters, or ten Java chars or 20 bytes per row. A VARCHAR2(4000) column will take 8K bytes per row. What matters is the defined size of the column, not the size of the actual data. A VARCHAR2(4000) column that contains all NULLs will still take 8K bytes per row. The buffers are allocated before the driver sees the query results so the driver must allocate enough memory for the largest possible result. A column defined as VARCHAR2(4000) can contain up to 4000 characters. The buffer must be allocated large enough to hold 4000 chars, even if the actual result data is not so large.

BFILE, BLOB, and CLOB values are stored as locators. The locators can be up to 4K bytes, so for each BFILE, BLOB, and CLOB column, the byte[] must have at least 4K bytes per row. RAW columns can contain up to 4K bytes. Other types take substantially less. A reasonable approximation is to assume 22 bytes per column per row for all other types.

Example:

```
CREATE TABLE TAB (ID NUMBER(10), NAME VARCHAR2(40), DOB DATE)
ResultSet r = stmt.executeQuery("SELECT * FROM TAB");
```

When the driver executes the executeQuery method, the database will parse the SQL. The database will report that the result will have three columns: a NUMBER(10), a VARCHAR2(40), and a DATE. The first column needs (approximately) 22 bytes per row. The second column needs 40 chars per row. The third column needs (approximately) 22 bytes per row. So one row needs 22 + (40 * 2) + 22 = 124 bytes per row. Remember that each character needs two bytes. The default fetchSize is 10 rows, so the driver will allocate a char[] of 10 * 40 = 400 chars (800 bytes) and a byte[] of 10 * (22 + 22) = 440 bytes, total of 1240 bytes. 1240 bytes is not going to cause a memory problem. Some query results are bigger.

In the worst case, consider a query that returns 255 VARCHAR2(4000) columns. Each column takes 8k bytes per row. Times 255 columns is 2040K bytes or 2MB per row. If the fetchSize is set to 1000 rows, then the driver will try to allocate a 2GB char[]. This would be bad.

The 12c JDBC drivers only allocate enough memory to store the query metadata and the actual query data. Instead of preallocating enough memory to store the maximum possible value for each column, the 12c drivers only allocate memory as needed to store the actual column values. This imposes some additional overhead but in most cases results in vastly less memory usage.

The 12c drivers allocate 15 bytes per value plus however much memory is required to store the actual data values. If a value is null, then only those 15 bytes are allocated. If the column is a VARCHAR2(32000) and a particular value is 32,000 characters, then 15 + 64,000 bytes (2 bytes per character) are allocated for that value. If the next value in the column is 1 character, then 15 + 2 bytes are allocated for that value.

This is a radical departure from the 10i and 11g approach. If all the values are maximum size, the 12c approach uses 13 bytes more per value than 10i and 11g. That is rarely the case. More typically there are many NULLs and VARCHAR values vary in size. Except in rare cases the 12c driver will use less memory than 10i or 11g.

The 12c drivers only use byte buffers. There are no char buffers. The buffers are all the same length. They are cached and reused. The buffer cache was designed in cooperation with the Java memory management team to optimize reuse while minimizing the overall memory footprint. It is worth noting that the 12c buffer size is larger than the minimum buffer size for 10i and 11g. For very small query results will use more memory, but the difference is just a few kilobytes.

Managing the buffer sizes

There are several things a user can do to manage the amount of memory used by these buffers.

1 Define tables carefully

2 Code queries carefully

3 Set the fetchSize carefully

Defining a column as VARCHAR2(4000) when VARCHAR2(20) would do makes a big difference to the 10i and 11g drivers. A VARCHAR2(4000) column requires 8K bytes per row. A VARCHAR2(20) column needs only 40 bytes per row. If the column in fact never hold more than 20 characters, then most of the buffer space allocated by the 10i or 11g driver for a VARCHAR2(4000) column is wasted. This has no impact on the 12c driver.

Doing SELECT * when only a few columns are needed has a big performance impact aside from just the buffer size. It takes time to fetch the row content, convert it, send it over the network, and convert it to the Java representation. Returning dozens of columns when only a few are needed forces the 10i and 11g drivers to allocate large buffers to store those unneeded results. It forces the 12c drivers to allocate storage for the actual column values. While this may be smaller than in 10i and 11g, it is still a waste. The 12c driver must allocate 15 bytes per value even for NULLs.

The primary tool for controlling memory use is the fetchSize. Although 2MB is pretty big, most Java environments wouldn't have any trouble allocating buffers that size. Even the Oracle database 11 worst case result of 255 VARCHAR2(4000) columns would not be a problem in most applications if the fetchSize were set to 1. The Oracle database 12 worst case of 1000 VARCHAR2(32000) would require the 10i and 11g drivers to allocate 64MB per row. Even with a fetchSize of 1, that might be a problem for many Java environments. The 12c driver would only allocate the memory necessary to store the actual data.

The first step in addressing a memory use issue is to review the SQL. Calculate the approximate size per row for each query and look at the fetch size. If the size per row is very large consider whether it would be possible to fetch fewer columns or to modify the schema to more tightly constrain the data sizes. Finally set the fetch size to keep the buffers to a reasonable size. What is “reasonable” depends on the details of the application. Oracle suggests the fetchSize be no more than 100, although in some cases larger size may be appropriate. A fetchSize of 100 may be inappropriately large for some queries even when many rows are returned.

Note: The Oracle specific method OracleStatement.defineColumnType may be used with the OCI driver to reduce the defined size of an overly large column. When called with a size argument, that size overrides the schema defined size for that column. This allows you to work around cases where you do not have the freedom to revise the schema to correct a problem. You must either not call defineColumnType at all on a given statement or call it for all columns for that statement. Fixing the schema is better when possible. The defineColumnType method is not supported in the 12c Thin driver except to set the LOB prefetch size for BLOB and CLOB columns.

One statement does not make a problem

Excluding the pathological cases like 1000 VARCHAR2(32000) columns or setFetchSize(100000), a single statement is unlikely to cause a memory use issue. In practice, problems only appear in systems that have hundreds or even thousands of Statement objects. A very large system might have a couple of hundred connections open simultaneously. Each connection might have one or two statements open at once. Such a large system would run on a machine with very large physical memory. With reasonable effort even such a very large system with a few hundred open statements is unlikely to have serious memory issues. Yes, the drivers would be using a lot of memory, but that's what memory is for, to be used. So in practice even very large systems can avoid memory problems.

Large systems tend to execute the same SQL many times. For performance reasons it is beneficial to reuse the PreparedStatements for each SQL rather than recreating them from scratch. So large systems can have many PreparedStatements, one (or more) for each distinct SQL string. Most large systems use a modular framework such as WebLogic Server. Independent components within the framework create the PreparedStatements they need. This conflicts with the need to keep PreparedStatements around. To address this need frameworks provide statement caching. With statement caching it is easy for each connection to have a hundred or more PreparedStatements in memory. Multiply by hundreds of connections and you have the potential for a real memory problem.

The Oracle JDBC drivers address this problem via the drivers' built in statement cache, the Implicit Statement Cache. (There is also an Explicit Statement Cache that is not discussed here.) The Implicit Statement Cache is transparent. The user code calls prepareStatement just like creating a new object. If the driver can satisfy the prepare call from the cache it does. If not it creates a new object. Semantically the user code cannot distinguish between a new and reused object. From a performance perspective, it is faster to return a statement from the cache than to create it anew. The first execution of the cached statement is faster since the driver and server can reuse much of the state from previous executions.

The Implicit Statement Cache knows the internal structure of Oracle JDBC PreparedStatements. As a result it can manage that structure for optimum performance. In particular it can manage the char[] and byte[] buffers. Different versions of the drivers manage these buffers in different ways as Oracle developed a better understanding of how different buffer management schemes performed in real world applications. The 10i and 11g JDBC drivers can manage the char[] and byte[] buffers only when PreparedStatements are returned to the Implicit Statement Cache. If the PreparedStatement is not closed, then the driver has no way to know that the statement will not be immediately reused and so cannot do anything to manage buffer use. The 12c drivers have a more dynamic buffer mechanism and return buffers to the cache when the ResultSet is closed. While Oracle still recommends using the Implicit Statement Cache, the 12c drivers will not have an excessive memory footprint when using other statement caches such a the one in WebLogic Server.

Statement Batching and Memory Use

The row data buffers are not the only large buffers the Oracle JDBC drivers can create. They can also create large buffers to send PreparedStatement parameters to the database. Applications generally read more data than they write and write smaller chunks of data at one time. As a result the parameter data buffers tend to be much smaller than the row data buffers. However, using (misusing) statement batching it is possible to force the drivers to create large buffers.

When the application calls PreparedStatement. setXXX to set a parameter value, the driver stores the value. This takes little memory; just a reference for arrays and Object types like String, 8 bytes for long and double, and 4 bytes for all others. When the PreparedStatement is executed, the driver must send those values to the database as SQL types, not Java types. The driver creates a byte[] and a char[] buffer. The parameter data is converted to SQL representation which is stored in the buffers. Then the driver ships the bytes across the network. Since the driver has the actual data size before the buffer is allocated, it can create a minimum size buffers. If a statement is executed multiple times, the driver tries to reuse the buffers. If the new data values need a larger byte[] or char[] buffer, a larger buffer is allocated. With any reasonable amount of memory, it would be unlikely to run out of memory executing a single statement. With statement batching however, things are different.

Statement batching executes a single DML statement many times in one operation. To do this the driver must send all the parameter values for all of the DML executes at once. This means the driver must convert all of the parameter data to SQL representation and store it into the buffers. The number of executes in a batch, the batch size, is analogous to the fetch size for queries. While it is unlikely that converting the parameters for a single statement execute could cause a memory problem, very large batch sizes can.

In practice this is a rare problem, and only shows up with unreasonable large batch sizes, tens of thousands. Calling executeBatch every few hundred rows should solve the problem.

The 12c driver addresses this problem by breaking the batch up if it gets too large. It should be able to handle many thousands of rows in a batch without running out of memory. On the other hand it may take several round trips to send all the rows rather than just one. Oracle still recommends calling executeBatch every few hundred rows as the size of the batch still matters.

Version Specific Memory Management

This section describes how various versions of the Oracle JDBC drivers manage buffer memory and how users can tune the driver for maximum performance.

Note: When discussing the details of memory management it is convenient to ignore the detail that the 10i and 11g driver have both char[] and byte[] buffers. In the remainder of this section, statements about generic "buffers" apply equally and independently to both char[] and byte[] buffers.

Although Java memory management is quite good, allocating large buffers is expensive. It is not the actual malloc cost. That is very fast. Instead the problem is the Java language requirement that all such buffers be zero filled. So not only must a large buffer be malloc'ed, it must also be zero filled. Zero filling requires touching every byte of the allocated buffer. Modern processors with their multilevel data caches do ok with small buffers. Zero filling a large buffer overruns the processor data caches and runs at memory speed, substantially less than the maximum speed of the processor. Performance testing has repeatedly shown that allocating buffers is a huge performance drag on the drivers. This has led to a struggle to balance the cost of allocating buffers with the memory footprint required to save buffers for reuse.

Oracle Database Release 10g Oracle JDBC Drivers

The initial 10g drivers have a naive approach memory management. They manage memory for maximum performance. When a PreparedStatement is first executed, the necessary byte[] and char[] buffers are allocated. That's it. The buffers are freed only when the PreparedStatement itself is freed. The Implicit Statement Cache does nothing with respect to managing the buffers. All of the PreparedStatements in the Implicit Statement Cache hold their allocated byte[] and char[] buffers ready for immediate reuse. As a result the only way to manage memory with this version of the drivers is with setFetchSize, careful design of the schema, and careful coding of SQL queries. The intial 10g driver is quite fast, but can have memory management issues, including OutOfMemoryExceptions.

Oracle Database Release 10.2.0.4 Oracle JDBC Drivers

The 10.2.0.4.0 drivers added a connection property to begin to address the memory management issues that appeared in the initial 10g drivers. This connection property is all or nothing. If set, returning a PreparedStatement to the Implicit Statement Cache frees the buffers. The buffers are reallocated when the statement is returned from the cache. This simple approach greatly reduces memory use but at a substantial performance cost. As described above, allocating buffers is expensive.

The connection property is

```
oracle.jdbc.freeMemoryOnEnterImplicitCache
```

Its value is a boolean string, “true” or “false”. If “true” the buffers are freed when a PreparedStatement is cached. If “false”, the default, the buffers are retained, as in the initial 10g drivers. The property can either be set as a System property via -D or as a connection property on the getConnection method. Note that setting freeMemoryOnEnterImplicitCache does not cause the parameter value buffers to be released, only the row data buffers.

Oracle Database Release 11.1.0.6.0 Oracle JDBC Drivers

The JDBC development team recognized that the all or nothing approach used in 10.2.0.4.0 was less than ideal. The 11.1.0.6.0 drivers have a more sophisticated approach to memory management. There were two goals, minimize the amount of unused memory and minimize the cost of allocating buffers. These drivers introduce an internal buffer cache for each Connection. When a PreparedStatement is returned to the Implicit Statement Cache its buffers are cached in a buffer cache. When a PreparedStatement is retrieved from the Implicit Statement Cache the buffers are retrieved from the buffer cache. As a result, PreparedStatements in the Implicit Statement Cache do not hold the large buffers and buffers are reused rather than being reallocated. This provides substantial performance improvement over the 10g drivers with and without freeMemoryOnEnterImplicitCache.

As noted in the introduction, the size of the buffers can vary widely, from zero to tens or even hundreds of megabytes. The 11.1.0.6.0 buffer cache is simple, too simple as it turns out. All cached buffers are the same size. Since a buffer can be used for any PreparedStatement in the Implicit Statement Cache, that size has to be big enough for the PreparedStatement with the largest demand. If only one statement at a time is in use, then there is only one buffer and that buffer is used by all PreparedStatements. For some or even most statements that buffer will be too large, but it will be the right size for at least one statement in the cache. If the pattern of PreparedStatement reuse is not too skewed, then it will be more performant to keep that large buffer around and reuse it constantly than to keep a smaller buffer and reallocate the large buffer as needed. If multiple statements are open at once and one of the PreparedStatements require an excessively large buffer there is a potential memory problem.

Consider an application where one PreparedStatement requires a 10MB buffer and the rest use much smaller buffers. So long as only one statement per connection at a time is in use and that the large PreparedStatement is reused sufficiently often, there is no problem. Each statement is assigned the single 10MB buffer when it is in use and returns the buffer to the buffer cache when the PreparedStatement is returned to the Implicit Statement Cache. The single 10MB buffer is allocated once and constantly reused by the various PreparedStatements. Now consider what happens if two PreparedStatements are open at once. Both must have buffers. Since any PreparedStatement may be assigned any buffer all buffers must be the same the same size, the maximum size. With two PreparedStatements open at one time both buffers must be 10MB. Opening a second PreparedStatement, even one that needs only tiny buffers requires allocating a second 10MB buffer. If three statements are open at once, a third 10MB buffer is allocated. In a large system with hundreds of connections and multiple hundreds of PreparedStatements open at once, using the largest buffer size for all PreparedStatements can lead to excessive memory use. This is obvious in retrospect, but the development team did not appreciate how big the problem could be and did not see it in internal testing. This issue can be minimized by appropriate schema design, SQL coding and choice of fetchSize.

Note that the 11.1.0.6.0 driver does not support freeMemoryOnEnterImplicitCache as buffers are always released by the PreparedStatement when it is put into the cache. The released buffers are stored in the internal buffer cache.

Oracle Database Release 11.1.0.7.0 Oracle JDBC Drivers

The 11.1.0.7.0 drivers introduce a connection property to address the large buffer problem. This property bounds the maximum size of buffer that will be saved in the buffer cache. All larger buffers are freed when the PreparedStatement is put in the Implicit Statement Cache and re-allocated when the PreparedStatement is retrieved from the cache. If most PreparedStatements require a modest size buffer, less than 100KB for example, but a few require much larger buffers, then setting the property to 110KB allows the frequently used small buffers to be reused without imposing the burden of allocating many maximum size buffers. Setting this property can improve performance and can even prevent OutOfMemoryExceptions.

The connection property is

```
oracle.jdbc.maxCachedBufferSize
```

Its value is an int string, e.g. “100000”. The default is Integer.MAX_VALUE. This is the maximum size for a buffer which will be stored in the internal buffer cache. A single size is used for both the char[] and byte[] buffers. The size is in chars for the char[] buffers and bytes for the byte[] buffers. It is the maximum buffer size, not a predefined size. If maxCachedBufferSize is set to 100KB but the largest buffer size less than 100KB is only 50KB, then the buffers in the buffer cache will be 50KB. Changes in the value of maxCachedBufferSize only make a difference in performance when they include or exclude char[] and byte[] buffers from the drivers' internal buffer cache. Huge changes, even of megabytes, may make no difference. Equally a change of 1 can make a difference when it causes the inclusion or exclusion of a PreparedStatement's buffers. This property may be set as a System property via -D or as a connection property via getConnection.

If you need to set maxCachedBufferSize, start by estimating the buffer sizes for the SQL queries that require the largest buffers. In the process you may find that by tuning the fetch size for these queries you can achieve the desired performance. Considering the frequency of execution and the size of the buffers, pick a size such that most statements can use cached buffers, but still small enough so that the Java runtime can support the number of buffers needed in order to minimize the frequency with which new buffers have to be allocated.

Some applications have large numbers of idle connections relative to the number of threads. An application may need to connect to any one of a large number of databases, but only to one at a time. If there is approximately one connection per thread to each database and many more databases than threads, then there are many more idle connections than threads. Since by default the buffer cache is per Connection, idle Connections result in unused buffers sitting in the buffer caches. This means a larger memory footprint than is necessary. While this is a relatively uncommon situation, it is not unknown.

This case is addressed by setting the connection property

```
oracle.jdbc.useThreadLocalBufferCache
```

The value of this property is a boolean string, “true” or “false”. The default is “false”. When this property is set to “true”, the buffer cache is stored in a ThreadLocal rather than in the Connection directly. If there are many fewer threads than Connections, then this will reduce the amount of memory used. The property may be set as a System property via -D or as a connection property via getConnection.

All Connections with useThreadLocalBufferCache = “true” share the same static ThreadLocal field and thus the same set of buffer caches. All Connections with useThreadLocalBufferCache = “true” must have the same value for maxCachedBufferSize if it is set. If a thread uses first one Connection and then another, the two Connections will have some indirect impact on each other's performance by the size and number of buffers used. Typically all Connections using the ThreadLocal buffer cache will be part of the same application so this should not matter. If one thread creates a statement and another thread closes the statement, the buffers will migrate from one ThreadLocal buffer cache to another and may not be reused. This is not recommended practice. If all statements are created in one thread and closed in another, then there would be no buffer reuse at all. Again, this is not recommended, but if that is the case in your app, don't set useThreadLocalBufferCache to “true”. It is possible to have some Connections use the ThreadLocal buffer cache and some the default per Connection buffer cache.

Oracle Database Release 11.2 Oracle JDBC Drivers

The 11.2 driver has a more sophisticated buffer cache than 11.1.0.7.0. This buffer cache has multiple buckets. All buffers in a bucket are of the same size and that size is predetermined.. When a PreparedStatement is executed for the first time the driver gets a buffer from the bucket holding the smallest size buffers that will hold the result. If there is no buffer in the bucket, the driver allocates a new buffer of the predefined size corresponding to the bucket. When a PreparedStatement is closed, the buffers are returned to their appropriate buckets.

Since buffers are used for a range of size requirements, the buffers are usually somewhat larger than the minimum required. The discrepancy is limited and in practice often has no impact.

The 11.2 drivers will always run in the same or less memory than the 11.1 or 10.2.0.4.0 drivers. Sometimes this is deceptive as the 11.2 driver will run with a heap size that is too small for adequate performance. Just because it will run with smaller memory doesn't mean it should be deployed that way. It is not unusual that setting -Xms to a large value greatly improves performance with the 11.2 driver. See the section on Controlling the Java Heap below.

The 11.2 drivers support maxCachedBufferSize but it is of less importance. In 11.1 correctly setting maxCachedBufferSize can be the difference between outstanding performance and OutOfMemoryExceptions. With the 11.2 driver setting maxCachedBufferSize can sometimes improve performance in very large systems with large statement caches and SQL with widely divergent buffer size requirements. In 11.2 the value of maxCachedBufferSize is interpreted as the log base 2 of the maximum buffer size. For example if maxCachedBufferSize is set to 20 the max size buffer that is cached is 2^{20} = 1048576. For backwards compatibility, values larger than 30 are interpreted as the actual size rather than log2 of the size, but using powers of 2 is recommended.

It is usually the case that setting maxCachedBufferSize to a reasonable value has no impact. If you need to set maxCachedBufferSize, start with 18. If you have to set the value to less than 16, you probably need more memory.

The 11.2 drivers use the same buffer cache and caching scheme for the parameter data buffers. When a PreparedStatement is put in the Implicit Statement Cache, the parameter data buffers are cached in the buffer cache. When a PreparedStatement is executed for the first time or for the first time after being retrieved from the Implicit Statement Cache, it gets parameter data buffers from the buffer cache. Typically the parameter data buffers are much smaller than the row data buffers, but these buffers can be large with very large batch sizes. The 11.2 drivers also use the buffer cache for other large byte[] and char[] buffers such are for Bfile, Blob, and Clob operations.

The 11.2 drivers support useThreadLocalBufferCache as well. Its function and the recommendations on when and how to use it are the same as for 11.1.0.7.0.

The 11.2 drivers also add a new property to enable the Implicit Statement Cache.

```
oracle.jdbc.implicitStatementCacheSize
```

The value of the property is an integer string, e.g. “100”. It is the initial size of the statement cache. Setting the property to a positive value enables the Implicit Statement Cache. The default is “0”. The property can be set as a System property via -D or as a connection property via getConnection. Calling OracleConnection.setStatementCacheSize and/or

OracleConnection.setImplicitCachingEnabled overrides the value of implicitStatementCacheSize. This property may make it easier to enable the Implicit Statement Cache in cases where enabling it in code is impractical.

Oracle Database Release 12.1.0.1.0 Oracle JDBC Drivers

As previously described, the 12.1.0.1.0 drivers use a different memory management scheme. The 12c drivers are insensitive to the defined size of the columns. A VARCHAR2(32000) column uses no more memory than an VARCHAR2(1) column for the same data. Just like the 10i and 11g drivers, the 12c drivers are sensitive to the number of columns in the result, the fetchSize, plus the size of the actual data. Oracle continues to recommend careful schema design as this can have an impact on other parts of the system. Careful query design, retrieving the minimum set of columns and rows, and careful choice of the fetchSize are as important as ever.

The 12c drivers have no options to control the size of the buffers or of the buffer cache.

Some applications have large numbers of idle connections relative to the number of threads. An application may need to connect to any one of a large number of databases, but only to one at a time. If there is approximately one connection per thread to each database, then there are many more idle connections than threads. Since by default the buffer cache is per Connection, idle Connections result in unused buffers sitting in the buffer caches. This means a larger memory footprint than is necessary. While this is a relatively uncommon situation, it is not unknown.

This case is addressed by setting the connection property

```
oracle.jdbc.useThreadLocalBufferCache
```

The value of this property is a boolean string, “true” or “false”. The default is “false”. When this property is set to “true”, the buffer cache is stored in a ThreadLocal rather than in the Connection directly. If there are fewer threads than Connections, then this will reduce the amount of memory used. The property may be set as a System property via -D or as a connection property via getConnection.

All Connections with useThreadLocalBufferCache = “true” share the same static ThreadLocal field and thus the same set of buffer caches. All Connections with useThreadLocalBufferCache = “true” must have the same value for maxCachedBufferSize if it is set. If a thread uses first one Connection and then another, the two Connections will have some indirect impact on each other's performance by the size and number of buffers used. Typically all Connections using the ThreadLocal buffer cache will be part of the same application so this should not matter. If one thread creates a statement and another thread closes the statement, the buffers will migrate from one ThreadLocal buffer cache to another. This is not recommended practice. If all statements are created in one thread and closed in another, then there would be no buffer reuse. Again, this is not recommended, but if that is the case in your app, don't set useThreadLocalBufferCache to “true”. It is possible to have some Connections use the ThreadLocal buffer cache and some the default per Connection buffer cache.

Because the 12c drivers have a more dynamic memory management it is often reasonable to fetch LONG and LONG RAW columns directly into memory. If your app has enough memory to store the entire value for a LONG or LONG RAW column you can set the oracle.jdbc.useFetchSizeWithLongColumn property. By default, if a query returns as LONG or LONG RAW column the driver sets the fetchSize to 1 and reads the LONG or LONG RAW values from the network only on demand. If you set this property the driver will use the defined fetchSize and read the entire LONG or LONG RAW into memory just like a VARCHAR or RAW. Since LONG and LONG RAW values can be huge, do not set this property unless you know the values will fit into memory. If you set it inappropriately the driver will likely get an OutOfMemoryException. The largest single value that can be read into memory is 2GB if you have the heap to store it.

The connection property is

```
oracle.jdbc.useFetchSizeWithLongColumns
```

Its value is a boolean string, “true” or “false”. If “true” LONG and LONG RAW values are read into memory just like VARCHAR and RAW values. If “false”, the default, including a LONG or LONG RAW column in a query will force the fetchSize to 1 and read the LONG and LONG RAW values from the network only on demand. The property can either be set as a System property via -D or as a connection property on the getConnection method. Note that this property exists in the 10i and 11g drivers but is much less useful.

Controlling the Java Heap

Tuning Java runtime memory use is a bit of a black art. The two most important options are -Xmx and -Xms. There are other parameters that vary by Java runtime version and OS. -Xmx sets the maximum heap size that the Java runtime will use. -Xms sets the initial heap size. The default values depend on the OS and the Java runtime version but roughly, the default for -Xmx is 64MB and default for -Xms is 1MB. 32 bit JVMs will support heaps up to 2GB. 64 bit JVMs will support even larger heaps. These options accept values given with “k”, “m”, and “g” suffixes to indicate kilo-, mega-, and gigabytes, e.g. -Xmx1G.

The Oracle JDBC drivers give the best performance when running with adequate heap size. For most applications increasing the application heap size up to some limit will improve performance. Increases beyond that size will make no difference. If the heap size is so large that the machine runs out of physical memory and heap memory is paged to secondary storage, performance will suffer badly. When setting the heap size, it is not sufficient to set just the maximum, -Xmx. The 11.2 driver in particular is not very aggressive about allocating memory. It frequently will not grow the heap beyond the minimum in which it can run. If you only set the maximum heap size, -Xmx, the 11.2 driver may never actually use enough memory to reach maximum performance even if -Xmx would allow it. If you also increase the minimum heap size, -Xms, the 11.2 drivers will make use of the additional memory and frequently give better performance.

When tuning application performance it is important to test with a fixed heap size, that is with both -Xmx and -Xms set and given the same value. Letting the JVM choose a heap size can obscure changes in performance that would be visible with a fixed heap size. Generally a production application should run with a fixed heap size as well. Setting the -server option will cause the JVM to be less aggressive about minimizing the heap size and so will provide some performance improvement. Setting -Xms appropriately will often provide additional performance beyond that provided by setting -server alone. Oracle recommends setting -server, -Xms, and -Xmx for server applications. Usually -Xms and -Xmx should be set to the same value. This applies to all Oracle JDBC drivers, 10i, 11g, and 12c.

Conclusion

The Oracle JDBC drivers use can use substantial amounts of memory to achieve maximum performance. If memory is constraining the performance of your application you can tune for maximum performance. Ideally you should have a reproducible test case that is representative of the real world application workload. By systematically changing the various knobs and running the test you can identify the optimum settings. For all versions of the Oracle JDBC drivers, enabling the implicit statement cache, designing the database schema appropriately, coding SQL queries carefully and setting the fetch size appropriately are the first steps. Next increase the Java heap size by setting -Xmx and -Xms to the largest practical size. If that gives acceptable performance you can try reducing -Xmx and -Xms, if a smaller heap is of any value. If performance with the largest practical heap is less than desired and you are using the 10i or 11g drivers, you should experiment with freeMemoryOnEnterImplictCache, or maxCachedBufferSize as appropriate. It should be clear from the design of your application whether you should set useThreadLocalBufferCache. If you need to eek out every last bit of performance, try tuning the Java garbage collector. (see http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html) In most cases setting -Xmx and -Xms is all that is required.

Oracle JDBC Memory Management
June 2013
Author: Douglas Surber

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment