Raising the fetch size, good or bad?

Exploring Memory Management in Oracle JDBC 12c
Trivadis is a market leader in IT consulting, system integration, solution engineering and the provision of IT services focusing on Oracle and Microsoft technologies in Switzerland, Germany, Austria and Denmark. We offer our services in the following strategic business fields:

- Business Intelligence
- Application Development
- Infrastructure Engineering
- Training

Trivadis Services takes over the interacting operation of your IT systems.
With over 600 specialists and IT experts in your region.

- 14 Trivadis branches and more than 600 employees
- 200 Service Level Agreements
- Over 4,000 training participants
- Research and development budget: CHF 5.0 million
- Financially self-supporting and sustainably profitable
- Experience from more than 1,900 projects per year at over 800 customers
About me

- Oracle Database Administrator since 2010
- Former Java Developer (up & including Java SE 5)
- Certifications including
  - Oracle Certified Master 11g
  - Red Hat Certified Systems Administrator RHEL 6
  - Sun Certified Developer for the Java 2 Platform
- Blog: http://recurrentnull.wordpress.com
- Twitter: @zkajdan
Agenda

- What is meant by fetch size, and why should you care?
- Memory allocation for result buffers: 11g vs. 12c
- Test Setup
- Results
- Conclusion / Recommendations
What is Meant by Fetch Size, and Why Should You Care?
Fetch Size - in a Nutshell

- Fetch size determines the number of rows retrieved from the database server in one network round trip
- Default in Oracle JDBC is 10
- Set on a (PreparedStatement): `stmt.setFetchSize(n);`
- Depending on network latency, increasing fetch size could (should!) allow for substantial performance gains
A True Story

It was just for your best...
Fetch Size – Digging Deeper: Under The Hood

CDB1>PDB1> SQL> select employee_id, first_name, last_name from employees;

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Steven</td>
<td>King</td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>Kochhar</td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>De Haan</td>
</tr>
<tr>
<td>103</td>
<td>Alexander</td>
<td>Hunold</td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Ernst</td>
</tr>
<tr>
<td>105</td>
<td>David</td>
<td>Austin</td>
</tr>
</tbody>
</table>

Fetch Size – Digging Deeper: Under The Hood


Source: Oracle Database Concepts 12.1, October 2014
DECLARE
    l_ename emp.ename%TYPE := 'SCOTT';
    l_empno emp.empno%TYPE;
    l_cursor INTEGER;
    l_retval INTEGER;
BEGIN
    l_cursor := dbms_sql.open_cursor;
    dbms_sql.parse(l_cursor, 'SELECT empno FROM emp WHERE ename = :ename', 1);
    dbms_sql.define_column(l_cursor, 1, l_empno);
    dbms_sql.bind_variable(l_cursor, ':ename', l_ename);
    l_retval := dbms_sql.execute(l_cursor);
    IF dbms_sql.fetch_rows(l_cursor) > 0 THEN
        dbms_sql.column_value(l_cursor, 1, l_empno);
        dbms_output.put_line(l_empno);
    END IF;
    dbms_sql.close_cursor(l_cursor);
END;
```
String getEmployees = "select firstname, lastname from employee";
PreparedStatement stmt = conn.prepareStatement(getEmployees);

List<Employee> employees = new ArrayList<>();
ResultSet rset = stmt.executeQuery();

while (rset.next()) {
    Employee employee = new Employee();
    employee.setFirstname(rset.getString(1));
    employee.setLastname(rset.getString(2));
    employees.add(employee);
}

rset.close();
stmt.close();
```

Where does the fetch size come in?
Fetch Size – Digging Deeper: Two meanings

- By fetching, we may mean two things:
  - Fetching rows from a cursor (requires server round trip)
  - Fetching rows from a result set (client side)
- This is easier so see with OCI.
Fetch Size – Digging Deeper: OCI

In OCI, the number of rows to be fetched from a cursor is set on the statement handle as its `OCI_ATTR_PREFETCH_ROWS` attribute:

```c
OCIAttrSet (stm, OCI_HTYPE_STMT, &rows, sizeof(rows),
           OCI_ATTR_PREFETCH_ROWS, err);
```

When the statement is executed, it may be told how many rows to fetch from this result set into a defined output variable using the `iters` parameter (position 4). The output variable has to be an array if `iters > 0`:

```c
OCIStrmtExecute(svc, stm, err, iters, 0, 0, 0, OCI_DEFAULT);
```
Fetch Size – Digging Deeper: OCI

- In case more results have been pre-fetched than are stored in the output variable, the rows are buffered during calls to `OCIStmtFetch2`.

- The above JDBC code is equivalent to having defined a scalar output variable and having set `iters` to 0 in `OCIStmtExecute`:

  ```java
  while (ret = OCIStmtFetch2(stm, err, 1, OCI_FETCH_NEXT, 0, OCI_DEFAULT) == OCI_SUCCESS) {
    //do something
  }
  ```

- It will pre-fetch rows from the cursor as configured on the Statement and fetch from the result set one row at a time, buffering the rest in memory.
Fetch Size – Digging Deeper: OCI

This is where CLIENT SIDE MEMORY comes in.

Which makes setting fetch size basically a tradeoff between PERFORMANCE and RESOURCE USAGE.
Memory Allocation for Result Buffers:
11g vs. 12c
Memory Allocation – 11g

- Each Statement object has two buffers:
  - One char[] : for (N)CHAR and (N)VARCHAR2 columns
  - One byte[] : for all other column types (NUMBER, DATE, ...
- Buffer sizes are determined at PARSE TIME, depending on column definition
- The actual size of the data returned does not matter!
Memory Allocation – 11g

Example:

```sql
CDB1.PDB1 SQL> CREATE TABLE allocs (id NUMBER, created DATE,
2       short_desc VARCHAR2(30), long_desc VARCHAR2(4000));
```

To retrieve 1 row from this table, the driver will allocate:

- 22 bytes for the NUMBER column (in the byte[])
- 22 bytes for the DATE column (in the byte[])
- 60 bytes for the VARCHAR2(30) column (in the char[])
- 8000 bytes for the VARCHAR2(4000) column (in the char[])
Memory Allocation – 11g

- Now imagine a table with 255 columns defined as VARCHAR2(4000):
- You would need to allocate $255 \times 8000$ bytes = \(~2\) MB per row
- Independent of whether actual values are shorter or even absent (null)!
- These \(~2\) MB are for 1 row only
- they need to be multiplied by the fetch size!
  - If fetch size = 10: \(~20\) MB
  - If fetch size = 100: \(~200\) MB
  - If fetch size = 1000: \(~2\ GB\)
Memory Allocation – 12c

- *Statement* objects have one byte[] buffer only (for all column types)
- Bytes are allocated based on actual column *values* (plus a fixed 15 bytes per column)
- Imagine same table with 255 VARCHAR2(4000) columns again
- Imagine 170/255 columns are not null, with an average length of 30 characters
- Now we need just ~5 kb of buffers to store one result row in the *Statement*
- And to retrieve several rows, now we need
  - If fetch size = 10: ~ 50 kb
  - If fetch size = 100: ~ 500 kb
  - If fetch size = 1000: ~ 5 MB
Aside: Why Did They Change It?

- Optional new maximum VARCHAR2 size in 12c: 32k (with max_string_size=extended)
- Internally stored as LOBs
- But formally defined as e.g. VARCHAR2(32767)
- Determining required buffer size at parse time, based on column definitions, is no longer feasible!
So

...
Test Setup
Test Setup

- Load tests using Swingbench (http://www.dominicgiles.com/swingbench.html)
- Custom Java transaction selects from big table with LARGE character columns (200 VARCHAR2(4000) columns, with an average length of 160 bytes each)
- Each transaction selects ~ 1000 rows, to allow for meaningful variations in fetch size
- Test run duration 20 minutes, 4 concurrent users
- DB version 12.1.0.2, client java version 1.8.0_45
- Maximum client heap size 1G
- Independent variable: JDBC driver version - 11.2.0.4 (ojdbc6.jar) vs. 12.1.0.2 (ojdbc6.jar)
Dependent Variables: Swingbench

Number of completed transactions

-TimeOfRun>Jun 14, 2015 8:27:25 AM</TimeOfRun>
<TotalRunTime>0:15:00</TotalRunTime>
<TotalLogonTime>0:00:03</TotalLogonTime>
<TotalCompletedTransactions>79</TotalCompletedTransactions>
<TotalFailedTransactions>0</TotalFailedTransactions>
<AverageTransactionsPerSecond>0.09</AverageTransactionsPerSecond>
<MaximumTransactionRate>8</MaximumTransactionRate>
<AverageResponse>45418.759493670885</AverageResponse>
<MinimumResponse>41665</MinimumResponse>
<MaximumResponse>48357</MaximumResponse>
<GeometricMean>45381.87265448191</GeometricMean>
<Skewness>-0.6101485966611743</Skewness>
<Kurtosis>-0.6940221560736153</Kurtosis>
<StdDeviation>1826.3569048735317</StdDeviation>
Dependent Variables: Java Flight Recorder

- Maximum and average heap sizes

![Heap Usage](image)

- Memory allocated for char[] and byte[] buffers

<table>
<thead>
<tr>
<th>Class</th>
<th>Instances</th>
<th>Size</th>
<th>Percentage of He</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>12,023</td>
<td>342.24 MB</td>
<td>99.75%</td>
</tr>
<tr>
<td>byte[]</td>
<td>537</td>
<td>231.34 kB</td>
<td>0.07%</td>
</tr>
</tbody>
</table>
**Dependent variables: OS memory**

- `/usr/bin/time`: OS level maximum memory usage (max resident size)

```
10.06user 14.16system 20:02.02elapsed 2%CPU (@avgtext+@avgdata 128676maxresident)k
32936inputs+592outputs (6major+78356minor)pagefaulpts 0swaps
```
Dependent variables: SQL Trace

Raw trace file

PARSING IN CURSOR #139782612622088 len=63 dep=0 uid=113 oct=3 lid=113 tim=2643223245 hv=1959287399 ad='fbf37a30' sqlid='ahuqtjxchqmi47' select * from mining_info where customer_id between :1 and :2
END OF STMT
PARSE #139782612622088: c=0,e=95,p=0,c=r=0,pu=0,mis=0,r=0,dep=0,og=1,plh=584152139,tim=2643223244
EXEC #139782612622088: c=100,e=2759,p=0,c=r=0,pu=0,mis=0,r=0,dep=0,og=1,plh=584152139,tim=2643226090
WAIT #139782612622088: nam='SQL*Net message to client' ela= 4 driver id=675562835 bytes=1 p=0 obj=0=1 tim=2643226224
WAIT #139782612622088: nam='Disk file operations I/O' ela= 17 FileOperation=2 fileno=18 filetype=2 obj#96222 tim=2643229263
WAIT #139782612622088: nam='Disk file operations I/O' ela= 2499 FileOperation=2 fileno=0 filetype=15 obj#96222 tim=2643231852
WAIT #139782612622088: nam='Disk file operations I/O' ela= 18 FileOperation=2 fileno=0 filetype=15 obj#96222 tim=2643233052
WAIT #139782612622088: nam='direct path read' ela= 6817 file number=18 first dba=484228 block cnt=124 obj#96222 tim=2643240321
WAIT #139782612622088: nam='Disk file operations I/O' ela= 23 FileOperation=2 fileno=0 filetype=15 obj#96222 tim=2643240579
WAIT #139782612622088: nam='Disk file operations I/O' ela= 12 FileOperation=2 fileno=0 filetype=15 obj#96222 tim=2643240826
WAIT #139782612622088: nam='direct path read' ela= 6567 file number=18 first dba=494482 block cnt=126 obj#96222 tim=2643247428

<nclip>

WAIT #139782612322088: nam='SQL*Net more data to client' ela= 4 driver id=675562835 bytes=8171 p=0 obj=0=96222 tim=2647681383
WAIT #139782612322088: nam='SQL*Net more data to client' ela= 4 driver id=675562835 bytes=8081 p=0 obj=0=96222 tim=2647681168
WAIT #139782612322088: nam='SQL*Net more data to client' ela= 5 driver id=675562835 bytes=8042 p=0 obj=0=96222 tim=2647681135
FETCH #139782612322088: c=1800,e=1055765,p=0,c=91,pu=0,mis=0,r=10,dep=0,og=1,plh=584152139,tim=2647681148
**Dependent variables: SQL Trace**

**TKPROF output**

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>40</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>40</td>
<td>0.00</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>3917</td>
<td>27.01</td>
<td>2973.19</td>
<td>2022034</td>
<td>1890904</td>
<td>0</td>
<td>38810</td>
</tr>
</tbody>
</table>

**Total**

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>3997</td>
<td>27.02</td>
<td>2973.20</td>
<td>2022034</td>
<td>1890904</td>
<td>0</td>
<td>38810</td>
</tr>
</tbody>
</table>

- SQL*Net message to client: 3917 0.00 0.01
- Disk file operations I/O: 20 0.00 0.01
- direct path read: 6925 0.03 33.40
- SQL*Net more data to client: 152113 1.17 2845.25
- SQL*Net message from client: 3917 0.75 1818.13
- db file sequential read: 302142 0.03 79.81
- resmgr:cpu quantum: 5 0.01 0.05
Results
11g: Completed Transactions

11.2.0.4 Throughput

Completed Transactions

Fetch Size

1 10 50 100 150 200 250 300 350 400 500

0 10 20 30 40 50 60 70 80 90
12c : Completed Transactions

12.1.0.2 Throughput

Completed Transactions

Fetch Size

1 10 50 100 150 200 250 300 350 400 500 1000 2000 5000 10000

33 11/19/2015 Raising the Fetch Size, Good or Bad?
Database Time

11.2.0.4, fetch size 1
- SQL*Net message from client
- FETCH: db file sequential read
- FETCH: direct path read
- FETCH: SQL*Net more data to client

11.2.0.4, fetch size 10
- SQL*Net message from client
- FETCH: db file sequential read
- FETCH: direct path read
- FETCH: SQL*Net more data to client
Database Time

11.2.0.4, fetch size 50
- SQL*Net message from client
- FETCH: db file sequential read
- FETCH: direct path read
- FETCH: SQL*Net more data to client

11.2.0.4, fetch size 100
- SQL*Net message from client
- FETCH: db file sequential read
- FETCH: direct path read
- FETCH: SQL*Net more data to client
Database Time

12.1.0.2, fetch size 1000
- SQL*Net message from client
- FETCH: db file sequential read
- FETCH: direct path read
- FETCH: SQL*Net more data to client

12.1.0.2, fetch size 10000
- SQL*Net message from client
- FETCH: db file sequential read
- FETCH: direct path read
- FETCH: SQL*Net more data to client
11g: Memory Usage

11.2.0.4 Heap Usage

- Maximum heap usage MB
- Average heap usage MB

fetch size

[Graph showing heap usage for different fetch sizes]
11g: Memory Usage

11.2.0.4 Memory Usage

- **Maximum heap usage MB**
- **Maximum OS memory usage MB**

Fetch size

```
0  200  400  600  800  1000  1200
1  10  50  100  150  200  250  300  350  400  500
```
11g : Buffers

11.2.0.4 char[] MB allocated

fetch size

char[] MB allocated
11g: Buffers

11.2.0.4 byte[] MB allocated

fetch size

byte[] MB allocated

1 10 50 100 150 200 250 300 350 400 500
12.1.0.2 Heap Usage

- Maximum heap usage MB
- Average heap usage MB

fetch size
12.1.0.2 Memory Usage

- Maximum heap usage MB
- Maximum OS memory usage MB

fetch size
12c: Buffers

12.1.0.2 char[] MB allocated

fetch size

char[] MB allocated
12c : Buffers

12.1.0.2 byte[] MB allocated

fetch size

byte[] MB allocated
Relating Throughput and Memory Usage: 11g

11.2.0.4: No. of Transactions vs. Memory Usage

- **fetch size**
- **char[] alloc’ed (MB)**
- **max heap (MB)**
- **No. transactions**

Graph showing the relationship between the number of transactions and memory usage for different fetch sizes.
Relating Throughput and Memory Usage: 12c

12.1.0.2: No. of Transactions vs. Memory Usage

- `byte[] alloc'ed (MB)`
- `max heap MB`
- `No. transactions`

fetch size

1 10 50 100 150 200 250 300 350 400 500 1000 2000 5000 10000

0 100 200 300 400 500 600 700 800 900
Conclusion / Recommendations
Conclusion

- Yes, new memory management in the 12c driver has a clear impact on possible throughput
- Chances are much bigger now that the optimal fetch size for a statement may be used
- Still, avoid configuring overly large fetch sizes (which anyway could not possibly do any good for most applications)
Recommendations – Version Independent

- Don’t fetch all columns (SELECT *) when only a few are needed
  - Especially not in combination with multiple joins
  - If you use an O/R Mapper check out if this is an option

- Find and configure the optimal fetch size for your application - possibly even for different statements / statement types

- Take into account your application’s display behavior – no need for a high fetch size if application pages through result in sets of 20 rows …

- Also keep in mind response time (in addition to throughput) – again, avoid excessive fetch sizes
And whereto from here?

- If you still need to economise on memory:
- Look into other memory and throughput related topics such as batching and statement caching
- But these already are topics for another time 😊
Questions?

Thank You!