**Improve Query Performance**

This section demonstrates a way to improve query performance by increasing the number of rows returned in each batch from Oracle to the Python program. Perform the following steps:

|  |
| --- |
| First, create a table with a large number of rows. Review the following [query\_arraysize.sql](http://www.oracle.com/technology/obe/11gr2_db_prod/appdev/opensrclang/python/files/query_arraysize.sql) script.  set echo on    drop table bigtab;    create table bigtab (mycol varchar2(20));    begin    for i in 1..20000    loop    insert into bigtab (mycol) values (dbms\_random.string('A',20));    end loop;    end;    /    show errors    commit;        In a terminal window use SQL\*Plus to run the script:  **sqlplus pythonhol/welcome@127.0.0.1/orcl     @query\_arraysize exit**  query-arraysize-gif |
| Review the code that is contained in the [query\_arraysize.py](http://www.oracle.com/technology/obe/11gr2_db_prod/appdev/opensrclang/python/files/query_arraysize.py) file in the $HOME directory.  import time  import cx\_Oracle  con = cx\_Oracle.connect('pythonhol/welcome@127.0.0.1/orcl')  start = time.time()  cur = con.cursor()  cur.arraysize = 100  cur.execute('select \* from bigtab')  res = cur.fetchall()  # print res # uncomment to display the query results  elapsed = (time.time() - start)  print elapsed, " seconds"  cur.close()  con.close()  This uses the 'time' module to measure elapsed time of the query. The arraysize is set to 100. This causes batches of 100 records at a time to be returned from the database to a cache in Python. This reduces the number of "roundtrips" made to the database, often reducing networks load and reducing the number of context switches on the database server. The fetchone(), fetchmany() and even fetchall() methods will read from the cache before requesting more data from the database.  From a terminal window, run:  **python query\_arraysize.py**  query-arraysize2-gif  Reload a few times to see the average times.  query-arraysize2-gif |
| Edit **query\_arraysize.py** and change the arraysize from  cur.arraysize = 100  to  cur.arraysize = 2000  Rerun the script a few times and compare the performance of the two arraysize settings. In general, larger array sizes improve performance. Depending how fast your system is, you may need to use different arraysizes than those given here to see a meaningful time difference.  python query\_arraysize.py  query-arraysize4-gif  The default arraysize used by cx\_Oracle is 50. There is a time/space tradeoff for increasing the arraysize. Larger arraysizes will require more memory in Python for buffering the records.  Using Bind Variables Bind variables enable you to re-execute statements with new values, without the overhead of reparsing the statement. Bind variables improve code reusability, and can reduce the risk of SQL Injection attacks.  To use bind variables in this example, perform the following steps.  .  .  .   |  | | --- | | Review the code as follows that is contained in the [bind\_query.py](http://www.oracle.com/technology/obe/11gr2_db_prod/appdev/opensrclang/python/files/bind_query.py) file in the $HOME directory.  import cx\_Oracle  con = cx\_Oracle.connect('pythonhol/welcome@127.0.0.1/orcl')  cur = con.cursor()  cur.prepare('select \* from departments where department\_id = :id')  cur.execute(None, {'id': 210})  res = cur.fetchall()  print res  cur.execute(None, {'id': 110})  res = cur.fetchall()  print res  cur.close()  con.close()  The statement contains a bind variable ":id". The statement is only prepared once but executed twice with different values for the WHERE clause.  The special symbol 'None' is used in place of the statement text argument to execute() because the prepare() method has already set the statement. The second argument to the execute() call is a Python Dictionary. In the first execute call, this associative array has the value 210 for the key of "id".  The first execute uses the value 210 for the query. The second execute uses the value 110.  From a terminal window, run:  python bind\_query.py  bind-query-gif  The output shows the details for the two departments. | | The cx\_Oracle driver supports array binds for INSERT statements, which can greatly improve performance over single row inserts.  Review the following commands to create a table for inserting data:  sqlplus pythonhol/welcome@127.0.0.1/orcl  drop table mytab;    create table mytab (id number, data varchar2(20));  exit    Run SQL\*Plus and cut-and-paste the commands.    bind-insert-gif | | Review the code as follows that is contained in the [bind\_insert.py](http://www.oracle.com/technology/obe/11gr2_db_prod/appdev/opensrclang/python/files/bind_insert.py) file in the $HOME directory.  import cx\_Oracle  con = cx\_Oracle.connect('pythonhol/welcome@127.0.0.1/orcl')  rows = [ (1, "First" ),  (2, "Second" ),  (3, "Third" ),  (4, "Fourth" ),  (5, "Fifth" ),  (6, "Sixth" ),  (7, "Seventh" ) ]  cur = con.cursor()  cur.bindarraysize = 7  cur.setinputsizes(int, 20)  cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)  #con.commit()  # Now query the results back  cur2 = con.cursor()  cur2.execute('select \* from mytab')  res = cur2.fetchall()  print res  cur.close() cur2.close() con.close()    The 'rows' array contains the data to be inserted.  The bindarraysize is here set to 7, meaning to insert all seven rows in one step. The setinputsizes() call describes the columns. The first column is integral. The second column has a maximum of 20 bytes.  The executemany() call inserts all seven rows.  The commit call is commented out, and does not execute.  The final part of the script queries the results back and displays them as a list of tuples.  From a terminal window, run:  **python bind\_insert.py**  bind-insert2-gif  The new results are automatically rolled back at the end of the script so re-running the script will always show the same number of rows in the table. |   **Creating Transactions**  When you manipulate data in an Oracle Database (insert, update, or delete data), the changed or new data is only available within your database session until it is committed to the database. When the changed data is committed to the database, it is then available to other users and sessions. This is a database transaction. Perform the following steps:  .   |  | | --- | | Edit the script used in the previous section bind\_insert.py and uncomment the commit call (in bold below):  import cx\_Oracle  con = cx\_Oracle.connect('pythonhol/welcome@127.0.0.1/orcl')  rows = [ (1, "First" ),  (2, "Second" ),  (3, "Third" ),  (4, "Fourth" ),  (5, "Fifth" ),  (6, "Sixth" ),  (7, "Seventh" ) ]  cur = con.cursor()  cur.bindarraysize = 7  cur.setinputsizes(int, 20)  cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)  con.commit()  # Now query the results back  cur2 = con.cursor()  cur2.execute('select \* from mytab')  res = cur2.fetchall()  print res  cur.close() cur2.close() con.close()    The commit() is on the connection, not on the cursor.  Rerun the script several times and see the number of rows in the table increasing each time:  **python bind\_insert.py**  bind-insert3-gif  If you need to initiate a rollback in a script, the con.rollback() method can be used.  In general you want all or none of your data committed. Doing your own transaction control has performance and data-integrity benefits. | |