Table of Contents

PanDA server migration from Oracle to MySQL DB backend....1

Introduction....2

PanDA server database backends....3

Class WrappedCursor....4

WrappedCursor implementation....4

Addressed issues....6

Table names in MySQL have to be case-insensitive....6

Schema names configured in panda_server.cfg....6

CURRENT_DATE term....6

SYSDATE term....6

EMPTY_CLOB() term....6

ROWNUM term....7

NOWAIT term....7

RETURNING INTO term....7

var of data type number....7

Sequences....7

Addressing sequence ATLAS_PANDA.JOBSDEFINED4_PANDAID_SEQ.nextval....9

Addressing sequence ATLAS_PANDA.CLOUDTASKS_ID_SEQ.nextval within the original table, with RETURNING INTO term....9

Database Schema....10

PanDA server migration from Oracle to MySQL DB backend

Introduction

- This page is evolving. Please keep checking.
- This page contains information about BigPanDA related migration of PanDA server from Oracle 11g to MySQL 5.1 performed in 2013-2014.
- Feel free to contact JaroslavaSchovancova and TadashiMaeno with questions.

PanDA server database backends

- PanDA server code is written in python, using Oracle as the main DB backend (e.g. for the ATLAS experiment). Most of the SQL queries in the code are written with Oracle's PL/SQL syntax.
- However, with a limited number of changes in the `panda_server.cfg` configuration file, one can decide whether Oracle or MySQL database backend will be used for PanDA data storage.
- `WrappedCursor()` is a class defined in `pandaserver.taskbuffer.WrappedCursor`. It alters Oracle's PL/SQL syntax, so that other DB backends can be used.
    - As of September 2014, the MySQL DB backend can be used.

Class WrappedCursor

- `WrappedCursor` class provides interface to translate Oracle PL/SQL queries to MySQL (or keep them as Oracle PL/SQL queries).
- Various Cursor methods are implemented.
- We use `cx_Oracle` and/or `MySQLdb` libaries for python-DB backend interactions.
  - http://cx-oracle.readthedocs.org/en/latest/cursor.html
  - http://mysql-python.sourceforge.net/MySQLdb.html#cursor-objects

WrappedCursor implementation

- available properties and methods of the WrappedCursor class

```
# connection object
conn = None

# cursor object
cur = None

# use special error codes for reconnection in querySQL
useOtherError = False

# backend
backend = 'oracle'

# constructor
def __init__(self, connection)

# iterator
def __iter__(self)

# string representation
def __str__(self)

def execute(self, sql, varDict=None, cur=None)

def executemany(self, sql, params)

def fetchall(self)

def fetchmany(self, arraysize=1000)

def fetchone(self)

def var(self, dataType, *args, **kwargs)

def getvalue(self,dataItem)

def next(self)

def close(self)

def prepare(self, statement)

# wrappers to handle Oracle's "RETURNING INTO" statement in different backends
def _returningIntoOracle(self, returningInputData, varDict, cur, dryRun=False)

def _returningIntoMySQLpre(self, returningInputData, varDict, cur)

def _returningIntoMySQLpost(self, returningInputData, varDict, cur)

@property
```

```
def description(self)

@property
def rowcount(self)

@property
def arraysize(self)

@arraysize.setter
def arraysize(self,val)
```

Addressed issues

- Here follows a list of issues that have been addressed in order to run PanDA server backed with MySQL from the same code as Oracle-backed version.

Table names in MySQL have to be case-insensitive

- Table names throughout the code are used with different capitalization. Therefore MySQL has to be configured so that table names are case-insensitive:

```
# cat /etc/my.cnf
[mysqld]
...
lower_case_table_names=1
...
```

Schema names configured in `panda_server.cfg`

- Oracle uses several schemas, while in MySQL a single schema is defined.
- In `WrappedCursor()` the schema names in a SQL query are replaced with schema names from configuration:

```
sql = re.sub('ATLAS_PANDA\.', panda_config.schemaPANDA + '.', sql)
sql = re.sub('ATLAS_PANDAMETA\.', panda_config.schemaMETA + '.', sql)
sql = re.sub('ATLAS_GRISLI\.', panda_config.schemaGRISLI + '.', sql)
sql = re.sub('ATLAS_PANDAARCH\.', panda_config.schemaPANDAARCH + '.', sql)
sql = re.sub('ATLAS_DEFT\.', panda_config.schemaJEDI + '.', sql)
```

`CURRENT_DATE` term

- Addressed in `WrappedCursor()`.
- Adapting Oracle's `CURRENT_DATE` for MySQL consists of 2 steps.
  - Digging the interval length.

```
sql = re.sub("CURRENT_DATE\s*-\s*(\d+|:[^\s\)]+)", "DATE_SUB(CURRENT_DAT
```

  - Using MySQL's `CURRENT_TIMESTAMP` instead of `CURRENT_DATE`, because of different date string formats in Oracle and MySQL.

```
sql = re.sub('CURRENT_DATE', 'CURRENT_TIMESTAMP', sql)
```

`SYSDATE` term

- Addressed in `WrappedCursor()`.
- Adapting Oracle's `SYSDATE` for MySQL consists of 2 steps.
  - Digging the interval length.

```
sql = re.sub("SYSDATE\s*-\s*(\d+|:[^\s\)]+)", "DATE_SUB(SYSDATE,INTERVAL
```

  - Using MySQL's `SYSDATE()` instead of `SYSDATE`. sql = re.sub('SYSDATE', 'SYSDATE()', sql)

`EMPTY_CLOB()` term

- Addressed in `WrappedCursor()`.
- There is no equivalent of `EMPTY_CLOB()` in MySQL, therefore replacing it with an empty string:

```
sql = re.sub('EMPTY_CLOB\(\)', "''", sql)
```

`ROWNUM` term

- Addressed in `WrappedCursor()`.
- Using MySQL's `LIMIT` instead of Oracle's `ROWNUM`.

```
sql = re.sub("(?i)(AND)*\s*ROWNUM\s*<=\s*(\d+)", " LIMIT \g<2>", sql)
sql = re.sub("(?i)(WHERE)\s*LIMIT\s*(\d+)", " LIMIT \g<2>" , sql)
```

`NOWAIT` term

- Addressed in `WrappedCursor()`.
- Oracle allows queries `SELECT ... FOR UPDATE NOWAIT`, however, MySQL does not support `NOWAIT`.
- In MySQL using empty string instead of `NOWAIT`.

```
sql = re.sub('NOWAIT', "", sql)
```

`RETURNING INTO` term

- Addressed in `WrappedCursor()`.
- In PanDA server the `RETURNING INTO` term is used to get an ID of the new job or file.
- In MySQL this functionality is replaced by `LAST_INSERT_ID()` together with an autoincremented column.

`var` of data type number

- Addressed in `pandaserver.taskbuffer.OraDBProxy`.
- When using `cursor.var()`, the Oracle-backed PanDA server returns variable of a data type `cx_Oracle.NUMBER`. However, `MySQLdb` provides no such thing, so we use python's `long` data type.

```
if panda_config.backend == 'oracle':
    import cx_Oracle
    varNUMBER = cx_Oracle.NUMBER
else:
    import MySQLdb
    varNUMBER = long
```

Sequences

- Addressed in `pandaserver.taskbuffer.OraDBProxy`, `pandaserver.taskbuffer.{JobSpec,FileSpec,DatasetSpec}`, `pandajedi.jedicore.{JediFileSpec,JediTaskSpec}`.
- Oracle provides sequences, while MySQL does not.
- However, MySQL provides autoincremented columns.
- In PanDA server `.nextval` or `.currval` properties of a defined sequence are used.
- In MySQL-backed server the sequence functionality is replaced by a definition of a special table as a counterpart for each sequence. The `.nextval` property is then replaced by `INSERT` of a `NULL` value into an autoincremented column of the "sequence table", combined together with `SELECT LAST_INSERT_ID()`.
- Adapting SQL quieries for sequences has to be taken into account also for the `JobSpec`, `FileSpec`, and `DatasetSpec` classes, e.g.

```
    # return expression of bind values for INSERT
```

```
    def bindValuesExpression(cls,useSeq=False):
        from config import panda_config
        ret = "VALUES("
        for attr in cls._attributes:
            if useSeq and cls._seqAttrMap.has_key(attr):
                if panda_config.backend == 'mysql':
                    # mysql
                    ret += "%s," % "NULL"
                else:
                    # oracle
                    ret += "%s," % cls._seqAttrMap[attr]
            else:
                ret += ":%s," % attr
        ret = ret[:-1]
        ret += ")"
        return ret
    bindValuesExpression = classmethod(bindValuesExpression)
```

- In MySQL-backed server the DB schema contains tables of the same name as the Oracle's sequence, e.g. `ATLAS_PANDA.JOBSDEFINED4_PANDAID_SEQ`.
  - The "sequence table" `JOBSDEFINED4_PANDAID_SEQ`:

```
CREATE TABLE JOBSDEFINED4_PANDAID_SEQ (
     id BIGINT(20) NOT NULL AUTO_INCREMENT,
     col bit(1) NULL,
     PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

- As of September 2014, the PanDA server uses the following sequences:

```
### ATLAS_PANDA schema ###
ATLAS_PANDA.JOBSDEFINED4_PANDAID_SEQ.nextval

ATLAS_PANDA.SUBCOUNTER_SUBID_SEQ.nextval

ATLAS_PANDA.GROUP_JOBID_SEQ.nextval

ATLAS_PANDA.CLOUDTASKS_ID_SEQ.nextval


### ATLAS_PANDAMETA schema ###

ATLAS_PANDAMETA.USERS_ID_SEQ.nextval

ATLAS_PANDAMETA.PROXYKEY_ID_SEQ.nextval

ATLAS_PANDAMETA.SITEACCESS_ID_SEQ.nextval


### ATLAS_DEFT schema ###

ATLAS_DEFT.PRODSYS2_TASK_ID_SEQ.nextval

ATLAS_DEFT.PRODSYS2_TASK_ID_SEQ.currval
```

- As of September 2014, the PanDA JEDI uses the following sequences:

```
### ATLAS_PANDA schema ###
ATLAS_PANDA.JEDI_DATASET_CONT_FILEID_SEQ.nextval

ATLAS_PANDA.JEDI_DATASETS_ID_SEQ.nextval

ATLAS_PANDA.JEDI_OUTPUT_TEMPLATE_ID_SEQ.nextval
```

Addressing sequence `ATLAS_PANDA.JOBSDEFINED4_PANDAID_SEQ.nextval`

- Code example:

```
    if panda_config.backend == 'mysql':
        ### fake sequence
        sql = " INSERT INTO ATLAS_PANDA.JOBSDEFINED4_PANDAID_SEQ (col) VALUES
        self.cur.arraysize = 10
        self.cur.execute(sql + comment, {})
        sql2 = """ SELECT LAST_INSERT_ID() """
        self.cur.execute(sql2 + comment, {})
        job.jobsetID, = self.cur.fetchone()
    else:  # panda_config.backend == 'oracle':
        sqlESS = "SELECT ATLAS_PANDA.JOBSDEFINED4_PANDAID_SEQ.nextval FROM dua
        self.cur.arraysize = 10
        self.cur.execute(sqlESS + comment, {})
        job.jobsetID, = self.cur.fetchone()
```

Addressing sequence `ATLAS_PANDA.CLOUDTASKS_ID_SEQ.nextval` within the original table, with `RETURNING INTO` term

- Code example:

```
if panda_config.backend == 'oracle':
    import cx_Oracle
    varNUMBER = cx_Oracle.NUMBER
else:
    import MySQLdb
    varNUMBER = long
### ...
if panda_config.backend == 'mysql':
    ### fake sequence
    sql = "INSERT INTO ATLAS_PANDA.cloudtasks (id,taskid,status,tmod,tenter) VA
else:  # panda_config.backend == 'oracle':
    sql = "INSERT INTO ATLAS_PANDA.cloudtasks (id,taskid,status,tmod,tenter) VA
sql += " RETURNING id INTO :newID"
varMap = {}
varMap[':taskid'] = cloudTask.taskid
varMap[':status'] = cloudTask.status
varMap[':newID']  = self.cur.var(varNUMBER)
self.cur.execute(sql+comment, varMap)
# get id
cloudTask.id = long(self.cur.getvalue(varMap[':newID']))
```

Database Schema

- `mysqldump -d` schema:
  https://svnweb.cern.ch/trac/panda/browser/bigpanda-misc/panda-DB-migration/mysql-work-misc/schema/sche
  - As of September 2014 it contains a plain list of tables' definitions, no indices.
  - Alternative for people without CERN account: guest view of the SVN:
    http://svn.cern.ch/guest/panda/bigpanda-misc/panda-DB-migration/mysql-work-misc/schema/schema.
- Schema columns description: PandaDB
- To be continued...

Major updates:

-- JaroslavaSchovancova - 09 Sep 2014

Responsible: JaroslavaSchovancova

This topic: PanDA > PandaServerOracleToMysqlMigration
Topic revision: r2 - 2014-09-10 - JaroslavaSchovancova