Optymalizacja baz danych

Report 2

Oskar Kwidziński

S156013

Informatyka Stosowana

Wydział Fizyki Technicznej i Matematyki Stosowanej

In accordance with commands to Report 2 a database test associated with multithreaded access has been created. Test was created in Python environment. Its main purpose is to simulate various aspects of three types of users (admins, heads of departments and cashiers). To do so a multithreaded access to database has been created. This allows to accomplish several tasks simultaneously, with the assuptions that any commit can concurre with any other at the same time. Especially, breaks during implementation of one task has been assumpted. Expected outcome of test is to check stability of called queries in dynamic conditions and measure time of command execution.

During test, access to database has been granted to three groups of users:

- 5 admins, with all system and objective permissions. Their task is to add records to tables 'Departments' and 'Products',
- 15 heads of departments, with objective permissions to update tables 'Employees' and 'Products',
- 35 cashier, that for purpose of the test are checking data in database with both complex and simple queries.

Script for realizing tasks mentioned above has been attached to the report. In this essay the main assuptions of test will be relieved.

```
class UserThread(Thread):
    available_methods = []

    def __init__(self, user_type, index):
        Thread.__init__(self)
        self.user_type = user_type
        self.index = index
        self.cursor = self._connect(user_type, index)

    @staticmethod
    def _connect(user_type, index):
        login = user_type + str(index)
        connection = cx_Oracle.connect(user=login, password=login, dsn=dsn_tns)
        return connection.cursor()

    def run(self):
        assert self.available_methods, 'You have to define at least one available method'
        timer = []
        print(self.getName(), 'has started')
        for i in range(0, executiontimes):
            time_start = time.time()
            if self.index%2:
                self.available_methods[0]()
            else:
                self.available_methods[1]()
            timer.append(time.time() - time_start)
            secondsToSleep = 7.52
            time.sleep(secondsToSleep)
        print('{}: finished, avarege time of execution: {}'.format(self.getName(), round(np.mean(timer), 4)))
```

Fig. 1. Class UserThread as a center of test execution.

Class showed if Fig. 1. is the main point of the test. It allows to connect all the users with '_connect' function, and then by 'cursor' method attached to 'cx_Oracle' proceed all 55 threads in the test. For any thread used, tasks given are repeated 100 times which is declared by 'executiontimes' parameter. However, for the purpose of test any method (task) given to thread is unchanged. It means that one command given to a certain thread is declared in advance. Between every compilation of thread a short time gap ('secondsToSleep' paramater) occurs. Its purpose is to simulate real traffic in database with commands and queries following one after another and breaks made by users.

```python
class Head(UserThread):

    def __init__(self, user_type, index):
        super().__init__(user_type, index)
        self.available_methods = [self.upd_empl, self.upd_prod]
        if self.index%2:
            self.setName('Thread HEAD{} updating employees'.format(index + 1))
        else:
            self.setName('Thread HEAD{} updating products'.format(index + 1))

    @timeit
    def upd_empl(self, *args, **kwargs):
        self.cursor.execute('''SELECT max(id) FROM sklepp.employees''')
        last_id = self.cursor.fetchone()[0]
        self.cursor.execute('''UPDATE sklepp.employees SET salary={} WHERE id={}'''.format(random.randint(2000,9000), random.randint(0, last_id)))
        self.cursor.execute('''COMMIT''')

    @timeit
    def upd_prod(self, *args, **kwargs):
        self.cursor.execute('''SELECT max(id) FROM sklepp.products''')
        last_id = self.cursor.fetchone()[0]
        self.cursor.execute('''UPDATE sklepp.products SET sell_price={} WHERE id={}'''.format(random.randint(1,20), random.randint(0, last_id)))
        self.cursor.execute('''COMMIT''')
```

Fig. 2. Class Head storing all the commands allowed for heads of departments users.

In Fig. 2. class 'Head' inheriting from previously shown class 'UserThread' has been presented. It explains that nearly half of the heads of departments are supposed to update peronal data of employees in 'Employees' table, while others are editing products specification (ex. price) in 'Products' table. Both aspects simulate real behaviour of an enterprise, where heads of departments are deciding about sell policy and salaries of employees. Both SQL commands have been passed to method 'cursor' and commited afterwards.

The two other types of users posses their own class with similiar complexity, but different in commands. 'Admin' class allows admins to insert data to certain tables (ex. new products sold or new employees) and 'Cashier' class allows cashiers to check for information in database (ex. finding in which departments work their colleagues that salary is higher then 5000 and order then by surname 😊). This approach allows to command the test in conditions in which data is constantly changing (due to previously explained functions 'upd_empl' and 'add_dept' a SQL query 'select …' is called).

```
Thread CASHIER33 selecting complex has started
Thread CASHIER34 selecting from random table has started
Thread CASHIER35 selecting complex has started
Thread CASHIER1 selecting complex: finished, avarege time of execution: 0.0018
Thread CASHIER2 selecting from random table: finished, avarege time of execution: 0.0019
Thread ADMIN1 adding records to products: finished, avarege time of execution: 0.0467
Thread ADMIN2 adding records to departments: finished, avarege time of execution: 0.0484
Thread HEAD2 updating employees: finished, avarege time of execution: 0.0855
Thread ADMIN2 adding records to departments: finished, avarege time of execution: 0.0491
Thread ADMIN4 adding records to departments: finished, avarege time of execution: 0.0421Thread ADMIN1
adding records to products: finished, avarege time of execution: 0.0558

Thread ADMIN3 adding records to products: finished, avarege time of execution: 0.0495
Thread ADMIN5 adding records to products: finished, avarege time of execution: 0.0661
Thread HEAD2 updating employees: finished, avarege time of execution: 0.0789
Thread HEAD6 updating employees: finished, avarege time of execution: 0.0555Thread HEAD4 updating
employees: finished, avarege time of execution: 0.0732

Thread HEAD8 updating employees: finished, avarege time of execution: 0.0567
Thread HEAD9 updating products: finished, avarege time of execution: 0.0666
Thread CASHIER1 selecting complex: finished, avarege time of execution: 0.004
Thread CASHIER2 selecting from random table: finished, avarege time of execution: 0.0023
Thread CASHIER3 selecting complex: finished, avarege time of execution: 0.0019
Thread HEAD10 updating employees: finished, avarege time of execution: 0.0829
```

Fig. 3. View from console after proceeding the test.

The outcome of the test allows to analize time execution of proceeded commands. As shown in Fig. 3. every specified thread have its time of execution measured and that allows to draw conclusions about about performance of database. In addition while using ‘timeit’ function it is possible to check every single time of executing a command. However, for the purpose of clarity of outcome shown it has been deactiveted.

Outcome shown above brings final conclusion about performance of database. It has been proofed that there is no dependency on number of users connected to database. Execution times during the test were similiar to that proceeded by only one user connected. However an issue associated with conccurency occured, especially when ‘secondsToSleep’ parameter was set to lower values. Conflict between updating and inserting data to the very same entity became factor which was throwing exeptions to the test and disallowed to proceed both commands simultaneously. To avoid this issue a ‘secondsToSleep’ parameter has been increased. Further works should take this issue into consideration as well as minimazing time execution for proceeded complex queries. Autoincrement implementation to entities and indexes performance in database may be a proposed solutions respectively.