Database Management Systems

Module 03: Why DBMS?/2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

Module Recap

- Evolution of Data and Records Management
- History of DBMS

Module Objectives

- Comparison of File based data management and DBMS

Module Outline

- File handling by Python viz-a-viz DBMS - Bank Transaction example
- Parameterized Comparison

Case Study of Bank Transaction

Banking Transaction System

Consider a simple banking system where a person can open a new account, transfer fund to an existing account and check the history of all her transactions till date.

The application performs the following checks:

- If the account balance is not enough, it will not allow the fund transfer
- If the account numbers are not correct, it will flash a message and terminate the transaction.
- If a transaction is successful, it prints a confirmation message.

Case study: A bank transaction (2)

We will use this banking transaction system to compare various features of a file-based (spreadsheet/.csv files) implementation viz-a-viz a DBMS-based implementation

- Account details are stored in
  - *Accounts.csv* for file-based implementation
  - *Accounts table* for DBMS implementation
- The transaction details are stored in
  - *Ledger.csv* file for file-based implementation
  - *Ledger table* for DBMS implementation

In the following slides we discuss a fund transfer transaction.

Source: *https://github.com/bhaskariitm/transition-from-files-to-db/tree/main*

Python viz-a-viz SQL

Bank Transaction: Python viz-a-viz SQL

Python

```
def begin_Transaction(creditAcc,
debitAcc, amount):

temp = []
success = 0

# Open file handles to retrieve and
    store transaction data

f_obj_Account1 =
    open('Accounts.csv', 'r')
f_reader1 =
    csv.DictReader(f_obj_Account1)
f_obj_Account2 =
    open('Accounts.csv', 'r')
f_reader2 =
    csv.DictReader(f_obj_Account2)
f_obj_Ledger =
    open('Ledger.csv', 'a+')
f_writer =
    csv.DictWriter(f_obj_Ledger,
    fieldnames=col_name_Ledger)
```

SQL

// Handled implicitly by the DBMS

Bank Transaction: Python viz-a-viz SQL (2)

Python

```
try:
  for sRec in f_reader1:
   #CONDITION CHECK FOR ENOUGH BALANCE
   if sRec["AcctNo"] == debitAcc and
   int(sRec["Balance"]) > int(amt):
...
```

SQL

```
do $$
begin
amt = 5000;
sendVal = '1800090';
recVal = '1800100';
select balance from accounts
into sbalance
where account_no = sendVal;
if sbalance < amt then
...
 $$
```

Bank Transaction: Python viz-a-viz SQL (3)

Python

```
try:
  for sRec in f_reader1:
   #CONDITION CHECK FOR ENOUGH BALANCE
   if sRec["AcctNo"] == debitAcc and
   int(sRec["Balance"]) > int(amt):
       for rRec in f_reader2:
           if rRec["AcctNo"] == creditAcc:
             sRec["Balance"] =                #DEBIT
                str(int(sRec["Balance"])-int(amt))
             temp.append(sRec)
...
```

SQL

```
do $$
begin
amt = 5000;
sendVal = '1800090';
recVal = '1800100';
select balance from accounts
into sbalance
where account_no = sendVal;
if sbalance < amt then
 raise notice "Insufficient balance";
else
update accounts
    set balance =
     balance - amt
    where account_no = sendVal;
...
$$
```

Bank Transaction: Python viz-a-viz SQL (4)

Python

```
try:
  for sRec in f_reader1:
   #CONDITION CHECK FOR ENOUGH BALANCE
   if sRec["AcctNo"] == debitAcc and
   int(sRec["Balance"]) > int(amt):
       for rRec in f_reader2:
           if rRec["AcctNo"] == creditAcc:
             sRec["Balance"] =                #DEBIT
                str(int(sRec["Balance"]) - int(amt))
             temp.append(sRec)
         # Critical point
             f_writer.writerow({
                "Acct1":sRec["AcctNo"],
                "Acct2":rRec["AcctNo"],
                "Amount":amt, "D/C":"D"})
             rRec["Balance"] =                #CREDIT
                str(int(rRec["Balance"]) + int(amt))
             temp.append(rRec)
...
```

SQL

```
do $$
begin
amt = 5000;
sendVal = '1800090';
recVal = '1800100';
select balance from accounts
into sbalance
where account_no = sendVal;
if sbalance < amt then
 raise notice "Insufficient balance";
else
update accounts
    set balance =
     balance - amt
    where account_no = sendVal;
insert into
  ledger(sendAc, recAc, amnt, ttype)
  values(sendVal, recVal, amt, 'D');
update accounts
    set balance =
     balance + amt
    where account_no = recVal;
...
```

Bank Transaction: Python viz-a-viz SQL (5)

Python

```
try:
  for sRec in f_reader1:
   #CONDITION CHECK FOR ENOUGH BALANCE
   if sRec["AcctNo"] == debitAcc and
   int(sRec["Balance"]) > int(amt):
       for rRec in f_reader2:
           if rRec["AcctNo"] == creditAcc:
             sRec["Balance"] =                #DEBIT
                    str(int(sRec["Balance"]) - int(amt))
             temp.append(sRec)
         # Critical point
             f_writer.writerow({"Acct1":sRec["AcctNo"],
                "Acct2":rRec["AcctNo"],
                "Amount":amt, "D/C":"D"})
             rRec["Balance"] =                #CREDIT
                str(int(rRec["Balance"]) + int(amt))
             temp.append(rRec)
             f_writer.writerow({"Acct1":rRec["AcctNo"],
                "Acct2":sRec["AcctNo"],
                "Amount":amt,"D/C": "C"})
             success = success + 1
             break
   f_obj_Account1.seek(0)
   next(f_obj_Account1)
   for record in f_reader1:
       if record["AcctNo"] != temp[0]["AcctNo"] and
            record["AcctNo"] != temp[1]["AcctNo"]:
                temp.append(record)
except:
   print("Wrong input entered !!!")
```

SQL

```
do $$
begin
amt = 5000;
sendVal = '1800090';
recVal = '1800100';
select balance from accounts
into sbalance
where account_no = sendVal;
if sbalance < amt then
 raise notice "Insufficient balance";
else
update accounts
    set balance =
     balance - amt
    where account_no = sendVal;
insert into
  ledger(sendAc, recAc, amnt, ttype)
  values(sendVal, recVal, amt, 'D');
update accounts
    set balance =
     balance + amt
    where account_no = recVal;
insert into
  ledger(sendAc, recAc, amnt, ttype)
  values(recVal, sendVal, amt, 'C');
commit;
raise notice "Successful";
end if;
end; $$
```

Bank Transaction: Python viz-a-viz SQL (6)

Python

```
#Writing back to the file

 f_obj_Account1.close()
 f_obj_Account2.close()
 f_obj_Ledger.close()

 if success == 1:
    f_obj_Account = open('Accounts.csv', 'w+', newline='')
    f_writer = csv.DictWriter(f_obj_Account,
        fieldnames=col_name_Account)
    f_writer.writeheader()
    for data in temp:
        f_writer.writerow(data)

    f_obj_Account.close()
    print("Transaction is successful !!")

 else:
    print('Transaction failed : Confirm Account details')
```

SQL

```
// Handled implicitly by the DBMS
```

Comparison

Parameter	File Handling via Python	DBMS
Scalability with respect to amount of data	Very difficult to handle insert, update and querying of records	In-built features to provide high scalability for a large number of records
Scalability with respect to changes in structure	Extremely difficult to change the structure of records as in the case of adding or removing attributes	Adding or removing attributes can be done seamlessly using simple SQL queries
Time of execution	In seconds	In milliseconds
Persistence	Data processed using temporary data structures have to be manually updated to the file	Data persistence is ensured via automatic, system induced mechanisms
Robustness	Ensuring robustness of data has to be done manually	Backup, recovery and restore need minimum manual intervention
Security	Difficult to implement in Python (Security at OS level)	User-specific access at database level
Programmer's productivity	Most file access operations involve extensive coding to ensure persistence, robustness and security of data	Standard and simple built-in queries reduce the effort involved in coding thereby increasing a programmer's throughput
Arithmetic operations	Easy to do arithmetic computations	Limited set of arithmetic operations are available
Costs	Low costs for hardware, software and human resources	High costs for hardware, software and human resources

Parameterized Comparison

Scalability

File handling via Python

- **Number of records:** As the # of records increases, the efficiency of flat files reduces:
  - the time spent in searching for the right records
  - the limitations of the OS in handling huge files
- **Structural Change:** To add an attribute, initializing the new attribute of each record with a default value has to be done by program. It is very difficult to detect and maintain relationships between entities if and when an attribute has to be removed.

DBMS

- **Number of records:** Databases are built to efficiently scale up when the # of records increase drastically.
  - In-built mechanisms, like indexing, for quick access of right data.
- **Structural Change:** During adding an attribute, a default value can be defined that holds for all existing records - the new attribute gets initialized with the default value. During deletion, constraints are used either not to allow the removal or ensure its safe removal

File handling via Python

- The effort needed to implement a file handler is quite less in Python
- In order to process a 1GB file, a program in Python would typically take few seconds.

DBMS

- The effort to install and configure a DB in a DB server is expensive & time consuming
- In order to process a 1GB file, an SQL query would typically take few milliseconds.

- If the number of records is very small, the overhead in installing and configuring a database will be much more than the time advantage obtained from executing the queries.
- However, if the number of records is really large, then the time required in the initialization process of a database will be negligible as compared to the time saved in using SQL queries.

File handling via Python

- **Persistence**: Data processed using in-memory data structures stay in the memory during processing. After updates, these are manually updated to the file on disk
- **Robustness**: Ensuring consistency, reliability and sanity is manual via multiple checks. On a system crash, a transaction may cause inconsistency or loss of data.
- **Security**: Extremely difficult to implement granular security in file systems. Authentication is at the OS level.

DBMS

- **Persistence**: Data persistence is ensured via automatic, system mechanisms. The programmer does not have to worry about the data getting lost due to manual errors
- **Robustness**: Backup, recovery & restore need minimum manual intervention. The backup and recovery plan can be devised for automatic recovery on a crash
- **Security**: DBMS provides user-specific access at the database level with restriction for to view only access

File handling via Python

- **Building the file handler**: Since the constraints within and across entities have to be enforced manually, the effort involved in building a file handling application is huge
- **Maintenance**: To maintain the consistency of data, one must regularly check for sanity of data and the relationships between entities during inserts, updates and deletes
- **Handling huge data**: As the data grows beyond the capacity of the file handler, more efforts are needed

DBMS

- **Configuring the database**: The installation and configuration of a database is specialized job of a DBA. A programmer, on the other hand, is saved the trouble
- **Maintenance**: DBMS has in-built mechanisms to ensure consistency and sanity of data being inserted, updated or deleted. The programmer does not need to do such checks
- **Handling huge data**: DBMS can handle even terabytes of data - Programmer does not have to worry

Arithmetic Operations

File handling via Python

- **Extensive support for arithmetic and logical operations**: Extensive arithmetic and logical operations can be performed on data using Python. These include complex numerical calculations and recursive computations.

DBMS

- **Limited support for arithmetic and logical operations**: SQL provides limited arithmetic and logical operations. Any other complex computation has to be done outside the SQL.

Costs and Complexity

File handling via Python

- File systems are cheaper to install and use. No specialized hardware, software or personnel are required to maintain filesystems.

DBMS

- Large databases are served by dedicated database servers need large storage and processing power
- DBMSs are expensive software that have to be installed and regularly updated
- Databases are inherently complex and need specialized people to work on it - like DBA
- The above factors lead to huge costs in implementing and maintaining database management systems

Module Summary

- Elucidated the difference between File handling by Python viz-a-viz DBMS through an Bank Transaction example
- Parameterized Comparison

Slides used in this presentation are borrowed from http://db-book.com/ with kind permission of the authors.
Edited and new slides are marked with "PPD".