12. Transaction Processing

Most of content here is borrowed from following sources-
>> Lecture slides of Jennifer Widom, Stanford (coursera)
>> Book Elmasri/Navathe

Transaction – Motivation

Following are two reasons of studying transactions.

1. Enable concurrent access of databases: DBMS receives data manipulate requests from multiple concurrent users. If we allow executing these requests in arbitrary order (or even in the order they arrive), it may either lead to wrong reporting, or more seriously may lead database becoming inconsistent.
2. Safeguard database against system failures - while databases are getting updated system might crash in between leaving database in inconsistent state. This problem is basically due to partial execution of a task on database.

We needs to have certain techniques implemented in DBMS to ensure that above two concerns are addressed. Here we study such techniques.

Transaction

What a "Transaction" is?

Consider relation model of databases; database operations are specifying using SQL. SQL is language to define operations on relations. However relational operations do not represent "business operations". Following are some examples of business tasks (operations)-

- Save an Order
- Transfer Fund from one account to another account
- Withdraw money
- Receive Money
- Give salary Raise
- Add a Customer

A business operation may require more than one SQL statements to be submitted to the DBMS. This is what brings in notion of Transaction. A transaction can be defined as a "sequence of one or more SQL statements" representing a business task.

While executing a business operation, that is a sequence of SQL statements (Transactions), we have two challenges-

1. We need to execute "operations" from multiple users concurrently, i.e. execution of concurrent requests from multiple users should go simultaneously; may be in "interleaved" fashion.
2. We do not want partial execution of any "Transaction"

Data Model for explaining transaction processing issues and techniques

We have "relations" at logical level, and have SQL to manipulate relations, and hence databases.

However "relation" data are stored on disks as "records". We perform lower level input/output operations for executing SQL statements. There are techniques to locate desired data records on disk blocks. For answering of queries, data from relevant blocks are brought into primary memory; may further process the data and present to the user.

For update operations, block(s) containing target data records are brought in the primary memory; necessary updates are done at appropriate places in blocks, and finally blocks are saved on disks at appropriate place aligned with corresponding file organization and access paths.

For transaction processing purposes, we use different data model, defined as following-

- A database is a collection of "named data items".
- Granularity of data can be a field, a record, a disk block, or even a whole file.
- Operations are just two "read" and "write"; and assume that all SQL statements are executed using these read/write primitives.

Let us also present an understanding of said "read" and "write" operations, as following-

mx = READ(data_x): reads database item **data_x,** and say it does following -

- Find block of **data_x** on disk
- Load block into buffer, if not already there
- Returns value of **data_x**

WRITE(data_x, mx): write memory value **mx** to **data_x** on database, and say it does following-

- Find address of **data_x** on disk
- Load block into buffer, if not already there
- Update **data_x** in buffer
- Write appropriate block into disk: may not be done immediately

For example consider following SQL statement-

```
UPDATE employee SET salary = salary + 3000 WHERE ssn = '1234';
```

In terms of said operations, let us say this SQL statement is executed as following

```
x = READ(SALARY1234);
x := x + 3000;
WRITE(SALARY1234 , x);
```

Let us take another example of a sequence of SQL statements expressed in terms of READ/WRITE statements. Let us say below is a transaction that transfers 3000 from acno 1011 to 5756-

```
UPDATE account SET balance = balance-3000 WHERE acno = 1011;
UPDATE account SET balance = balance+3000 WHERE acno = 5756;
```

The transaction expressed in terms of read/write operations:

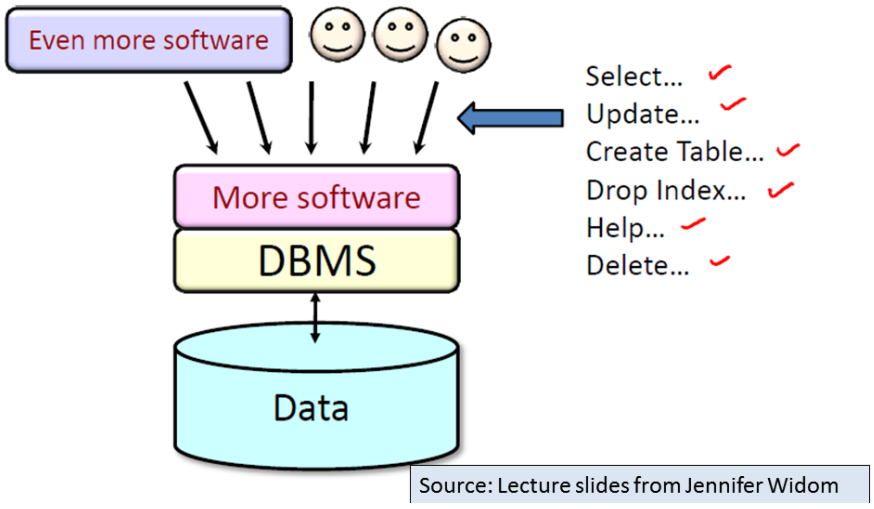
```
b = READ(BALANCE1011);
b := b - 3000;
WRITE(BALANCE1011, b);
b = READ(BALANCE5756);
b := b + 3000;
WRITE(BALANCE5756, b);
```

More operations on a transaction

1. BEGIN TRANSACTION (Indicates that transaction begins)
2. COMMITT TRANSACTION (Make the transaction updates permanent on disk files)
3. ROLLBACK TRANSACTION (Undo the updates of a transaction)

Problems due to concurrent Access

Any real database is accessed by multiple users; and in a typical OLTP environment, some users are reading data (answering select queries), and some clients are updating database. Diagram below should capture the scenario.



The process of concurrent execution of Transactions can be understood as following-

1. Let multiple users concurrently submit transactions to DBMS
2. Let each transaction SQL operation be translated into appropriate read/write operations (as discussed above) and queued to be executed by DBMS
3. Let DBMS include operations from multiple transactions and execute in interleaved serial order. Below is an example depicting interleaved "execution schedule" of DBMS-

   Suppose following three transactions are executing concurrently- T1(s1, s2, s3, s4), T2(s8, s9, s10), T3(s11, s12, s13, s14); and typically operation may be interleaved as following-

   ..., s1,s2,s8,s11,s9,s12,s3,s4,s13,s10,s14,...

Note that while interleaving statements from multiple transactions, order of statements from a transaction does not change, however instructions from other transaction get inserted in between.

Let us see some contexts to demonstrate problems arising due to concurrent access of databases. Take a note of relations given below, that will be used in following discussions-

- Employee(SSN, Name, Salary, DNO)
- JobApplications(AppNo, ExpectedSalary, HireScore, Experience, SalaryOffered, Hired)
- Account(AccNo, Balance)
- Transaction(AcNo, TS, Description, CrDr, Amuont)

Case-1

Say following two SQL statements are simultaneously submitted by two clients C1 and C2; both are updating balance of account no 1234. Let us say current balance is 10000.

C1: UPDATE account SET balance = balance + 3000 WHERE accno = 1234;
C2: UPDATE account SET balance = balance - 5000 WHERE accno = 1234;

Let us say, DBMS executes them in interleaved fashion in order as given below –

[C1]:	$x=READ(BALANCE_{1234})$	[x=10000]
[C2]:	$y=READ(BALANCE_{1234})$	[y=10000]
[C1]:	x := x + 3000	[x=13000]
[C2]:	y := y - 5000	[y=5000]
[C2]:	$WRITE(BALANCE_{1234},y)$	[C2 updates balance to 5000 in DB]
[C1]:	$WRITE(BALANCE_{1234},x)$	[C1 updates balance to 13000 in DB]

Correct update should be: [balance=8000]
Note the problem? Update of one transaction lost ==> LOST UPDATE problem

Case-2

Say following two statements are simultaneously submitted by two clients C1 and C2;

C1: UPDATE employee SET salary=60000 WHERE ssn = 1234;
C2: UPDATE employee SET dno=4 WHERE ssn = 1234;

Both clients are updating different attributes. If attribute level read/write is possible then two clients are accessing different attributes and there is no concurrency issue. However attribute level update is too low granularity in database manipulations, and let us say it is done at record/tuple level. Let us say execution schedule generated is as following –

[C1]:	$x = READ(EMP_{1234})$	[x=<1234, Anil, 3, 50000>]
[C1]:	x.salary := 60000	[x=<1234, Anil, 3, 60000>]
[C2]:	$y=READ(EMP_{1234})$	[y=<1234, Anil, 3, 50000>]
[C2]:	y.DNO=4	[y=<1234, Anil, 4, 50000>]
[C1]:	$WRITE(EMP_{1234},x)$	[<1234, Anil, 3, 60000>] //the tuple in DB
[C2]:	$WRITE(EMP_{1234},y)$	[<1234, Anil, 4, 50000>] //the tuple in DB

Correct update should be: [<1234, Anil, 4, 60000>]

Note the problem? What should actual state of the tuples in correct processing?
Update of one transaction lost ==> LOST UPDATE problem

Case-3

Say following two statements are simultaneously submitted by two clients C1 and C2;

```
C1:   UPDATE employee SET salary=60000 WHERE ssn = 1234;
      UPDATE employee SET DNO=3 WHERE ssn = 2345;
      ROLLBACK;

C2:   UPDATE employee SET salary=salary+5000 WHERE ssn = 1234;
```

In this case let us say DBMS is reading whole record (tuple), reading an attribute, let us say too a low granularity in database operations –

```
[C1]:  x = READ(EMP1234)            [x=<1234, Anil, 3, 50000>]
[C1]:  x.salary := 60000            [x=<1234, Anil, 3, 60000>]
[C1]:  WRITE (EMP1234,x)            [DB tuple: <1234, Anil, 3, 60000>]
[C2]:  y=READ(EMP1234)              [y=<1234, Anil, 3, 60000>]
[C2]:  y.salary := y.salary+5000    [y=<1234, Anil, 3, 65000>]
[C1]:  z = READ(EMP2345)            [z=<2345, Mukesh, 2, 40000>]
[C1]:  z.DNO := 3                   [z=<2345, Mukesh, 3, 40000>]
[C1]:  ROLLBACK     [DB tuples: <1234, Anil, 3, 50000>], [<2345, Mukesh, 2, 40000>]
[C2]:  WRITE (EMP1234,y)            [DB tuple: <1234, Anil,3, 65000>]
[C2]:  COMMIT
```

Correct update should be: [<1234, Anil, 3, 55000>]

Note the problem? What should actual state of the tuples in correct processing?
==> DIRTY READ

Case-4

Say following two statements are simultaneously submitted by two clients C1 and C2;

```
C1: UPDATE salary SET salary = 1.1*salary;
C2: SELECT avg(salary) FROM employee;
```

Say above SQL statements are translated into read/write and executed in interleaved manner as following [this translation is indicative, may not actually done this way]

```
C1:   for each employee e from EMP
            e = READ(EMP_e);    //reads a tuple
            e.salary := 1.1 * e.salary;
            WRITE(EMP_e, e);    //write the tuple
C2:   sum=0; n=0;
      for each employee e from EMP
            e = READ(EMP_e);    //read a tuple
            if ( e.salary <> null )
                  sum += e.salary;
                  n += 1;
      avg = sum / n;
```

Note the problem? INCONSISTENT READ;
C2 either should have read all old salary or all new salary!

Issue - System Failures

Suppose, we need to transfer amount of 5000 from account number 1011 to 2312; and we have following database updates to log the transaction and update the balances accordingly.

```
INSERT INTO transaction VALUES (1011, now, 'Transferred to A/c 2312','Debit', 5000);
INSERT INTO transaction VALUES (2312, now, 'Transferred from A/c 1011','Credit', 5000);
UPDATE account SET balance=balance–5000 WHERE acno = 1011;
UPDATE account SET balance=balance+5000 WHERE acno = 2312;
```

What if system crashes after executing 3rd statement? We ought to execute either all statements or none?

Conclusion drawn from issues

- Uncontrolled sequencing of operations for execution may bring database in inconsistent state!!
- Partial execution of a set of statements may bring database in inconsistent state.

Solution:

- Control the concurrent access: Order the Read/Write operations from multiple clients such that they appear to be running in isolation
- Guarantee executing sequence of related operations (Transactions) from a client in "all or nothing" manner, regardless of failures

ACID properties of Transaction

ACID is acronym for

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

These are desirable characteristics of Transaction Processing. If ensured, concurrent execution of transaction would not have any undesirable consequences.

ACID – ISOLATION

- If we execute transaction in isolation that is without interleaving, that means let DBMS finish one transaction before taking up turn of another.
- However we cannot afford such a isolation (i.e. without interleaving). This leads we having transaction going into longer wait periods, processor remaining under loaded, infinite loops in one transaction would actually halt the whole system, and so on.

This conflicting situation (we require isolation but we cannot afford it) brings in notion of **Serializability**. It defines a concept of interleaving schedule such that its result is equivalent to some **serial** (sequential) **order**.

So basically Isolation property requires that transaction execution should ensure serializability.

ACID – Durability

- This characteristic requires that if system crashes after transaction commits; all effects of transaction reflects in database.
- In real systems, DBMS might acknowledge the client for a commit request, but it may only be in buffer and actual update on disk may not occur immediately.
- When some committed updates are only in buffer, and system fails at this point of time, content of buffer may get lost. This implies that committed updates of transaction did not reflect in DBMS, and this is undesirable.
- Durability characteristic requires, "commits are durable" irrespective of system failures.

ACID - Atomicity

- A transaction is executed in "all or nothing" manner.
- A transaction is never half executed.
- To ensure this, it may be undesirable to write anything on the disk until transaction commits; however this may not be feasible. Buffer management techniques needs to flush changes to the disk when memory is not sufficient.
- Failure at this stage might leave a transaction half done on DBMS.

Durability and Atomicity are ensured by DBMS maintaining "logs of all updates" on database; and Durability and Atomicity is ensured by "redoing" certain operations (typically of committed transaction) and from logs. To ensure atomicity again systems uses logs and undoes typically undoes operations of uncommitted transactions.

ACID – Consistency

- Each transaction from each client can assume that all constraints hold when transaction begins.
- Transaction execution should ensure that all constraints hold true after transaction commits.
- Serializability and atomicity takes care of this property.