DBMS IMPLEMENTATION
CONCURRENCY CONTROL

Transaction

- transaction is the basic unit of work in a DBMS

4 basic or *acid* properties of a transaction:

atomicity
consistency
independency
durability

Concurrency Control Properties

- ***atomicity***
  the <u>all or nothing</u> property; a transactions is an indivisible unit of work
- ***consistency***
  transactions transform the DB from one consistent state to another consistent state
- ***independence***
  transactions execute independently of one another i.e. partial effects of one transaction are not visible to other transactions
- ***durability*** (also called ***persistence***)
  the effects of a successfully completed (committed) transaction are permanently recorded in the DB and cannot be undone

DB Processing

- Execution of application processing consists of a series of atomic transactions with non-DB processing taking place in between

Figure 6.1 - Program executing several transactions

DB Processing (contd.)

- ***transaction manager*** oversees execution of a transactions and co-ordinates DB requests on behalf of the transaction
- ***scheduler*** implements a particular strategy for transaction execution
- objective of the scheduler is to maximise concurrency without allowing transactions to interfere with one another and compromise the integrity of the DB

DB Processing (contd.)

Transactions can be interleaved in 2 ways:

- end-to-end txn execution Figure 6.2

- concurrent txn execution Figure 6.3

DB Processing (contd.)

- start of a transactions signalled by *begin transaction*

- end of a transaction signalled by either
  commit (= successful termination)
  or
  abort (= unsuccessful termination)
  (rollback)

Example DB Transaction

Figure 6.4 *Funds transfer transaction*

Example DB Transaction (contd.)

- [redacted]

- [redacted]

- [redacted]

Transaction Interference

There are three different ways in which concurrently executing transactions can interfere with one another:

- lost update problem
- violation of integrity constraints
- inconsistent retrieval

Lost update Problem

- Apparently successful updates can be overwritten by other transactions

Figure 6.5 Lost Update Problem

Violation of Integrity Constraints

Example DB:

SCHEDULE (surgeon_name, operation, date)

SURGEON (surgeon_name, operation)

- SCHEDULE specifies which surgeon is to perform a particular operation on a particular day.
- SURGEON records the qualifications by operation for eachoperation
- integrity constraint:

surgeons must be qualified to perform
the operations for which they are scheduled

Figure 6.6 shows the initial state of the DB,

Figure 6.7 shows the interleaving of 2 transactions, T_3 and T_4

Figure 6.6 Initial DB State

Figure 6.7 Interleaving of Txns T_3 and T_4

Violation of Integrity Constraints (Contd.)

- Figure 6.8 shows the invalid state of the DB after execution of T_3 and T_4 .

Figure 6.8

Note: Neither transaction is aware of the action of the other transaction as they are both updating different data

Inconsistent retrieval (dirty read)

- Most Concurrency Control work concentrates on transactions which update DB since only they can corrupt the DB
- If transactions are allowed to *read* the partial results of incomplete transactions, they can obtain an inconsistent view of the DB (*dirty or unrepeatable read)*
- Example given in Figure 6.9

Example: Inconsistent Retrieval Problem

Figure 6.9

Schedules and serialisation

- A transaction consists of a sequence of reads and writes to the database
- The entire sequence of reads and writes by all concurrent transactions in a database taken together is known as a ***schedule***.

Schedules and serialisation contd.

- a schedule *S* is generally written:

  $S = [\ O^1,\ O^2,\ O^3,\ \ldots\ldots\ldots\ldots,\ O^m]$

  where O^i indicates either a read (R) or write (W) operation executed by a transaction on a data item

- O^1 precedes O^2, which in turn precedes O^3, and so on. This is generally denoted

  $O^1 < O^2 < O^3 < \ldots\ldots\ldots\ldots < O^m$

Schedules and serialisation contd

- The schedule *S* for transactions T_1 and T_2 in Figure 6.5 would be

  $S = [R_2(\text{balance}_x), R_1(\text{balance}_x), W_2(\text{balance}_x), W_1(\text{balance}_x), R_1(\text{balance}_y), W_1(\text{balance}_y)]$

  where R_i and W_i denote read and write operations, respectively, by transaction T_i.

Schedules and serialisation contd

- Most concurrency control algorithms assume that transactions read a data item *before* they update it *(constrained write rule)* i.e.

  $R_i\ (x_j) < W_i\ (x_j)$.

- The order of interleaving of operations from different transactions is crucial to maintaining the consistency of the database

Serial schedule

A ***serial*** schedule is one in which all the reads and writes of each transaction are grouped together so that the transactions are run sequentially one after the other, as in Figure 6.2(a)

- Schedule, *S*, is said to be ***serialisable*** if all the reads and writes of each transaction *can* be reordered in such a way that when they are grouped together as in a serial schedule, the net effect of executing this serial schedule is the *same* as that of the original schedule *S*

Serial schedule contd.

- This reorganised schedule is called the *equivalent serial schedule* of the original *serialisable* schedule
- A serialisable schedule will therefore be *equivalent* to and have the same effect on the DB as *some* serial schedule

Example

S_1 = [$R_7(x)$, $R_8(x)$, $W_8(x)$, $R_6(y)$, $W_6(y)$, $R_7(y)$, $W_7(y)$]

Serial schedule contd.

- chronologically, by start-time, the order of execution of transactions is T_7, then T_8 and followed by T_6
- logically however T_6 precedes T_7 which in turn precedes T_8, since T_6 reads the pre-T_7 value of y, while T_7 sees the pre-T_8 value of x,

  i.e. $T_6 < T_7 < T_8$.

  in spite of the fact that chronologically T_8 finishes before T_6 begins!

Serial schedule contd.

- schedule S_1 for Figure 6.10 is

  S_1 = [$R_7(x)$, $R_8(x)$, $W_8(x)$, $R_6(y)$, $W_6(y)$, $R_7(y)$, $W_7(y)$]

and the equivalent serial schedule, SR_1 is

SR_1 = [$R_6(y)$, $W_6(y)$, $R_7(x)$, $R_7(y)$, $W_7(y)$, $R_8(x)$, $W_8(x)$]

Serial schedule contd.

- note that a serialisable schedule is *not* the same as a serial schedule
- *Serialisability is taken as proof of correctness since a serial schedule cannot contain transactions which interfere with each other and a serialisable schedule is a schedule which has the same net effect as executing a serial schedule*

=> one possible approach is to examine the schedule produced and see if it is serialisable

Serial schedule contd.

- However, deciding whether a schedule is equivalent to some serial schedule is a very difficult computational problem (with *constrained write* rule: polynomial complexity; without: NP-complete problem)
- better to design schedulers in such a way that they are *guaranteed* to generate only serialisable and hence correct schedules

Serial schedule contd.

Formally, we can state the rules for equivalence of schedules as:

Rule 1: Each read operation reads the same values in both schedules; this effectively means that those values must have been produced by the same write operations in both schedules

Rule 2: The final database state is the same for both schedules; thus the final write operation on each data item is the same in both schedules.

Conflicting operations

- Read operations cannot conflict with one another and the order of execution of $R_1(x)$ and $R_2(x)$ does not matter, i.e.

$$[R_1(x), R_2(x)] \; \circ \; [R_2(x), R_1(x)]$$

but

$$[R_1(x), W_1(x), R_2(x)] \; {}^{1} \; [R_1(x), R_2(x), W_1(x)]$$

Conflicting operations contd.

- In terms then of schedule equivalence, it is the ordering of *conflicting* operations which must be the same in both schedules
- The conflict between a read and a write operation is called a *read-write conflict*, and a conflict between two write operations a *write-write conflict*.

CONCURRENCY CONTROL TECHNIQUES

Three basic concurrency control techniques:

- locking methods (conservative)
- timestamp methods (conservative)
- optimistic methods

Locking methods

- Most widely used approach
- A transaction must claim a *read (shared*) or *write (exclusive*) *lock* on a data item prior to the execution of the corresponding read or write operation on that data item
- More than one transaction can hold read locks simultaneously on the same data item

Locking methods contd.

- Only one transaction at a time can hold a write lock on a data item
- Transaction holds a lock until it explicitly releases it
- Effects of the write operation are not visible to other transactions until the write lock has been released

Locking methods contd.

- in Figure 6.10, the request by transaction T1 for a write lock on $balance_x$ would be denied since T_5 already holds a read lock on $balance_x$.
- some systems allow **upgrading** and **downgrading** of locks
  i.e. a read lock on a data item can be *upgraded* to a write lock, if it is the *only* transaction holding a read lock on that data item; similarly a write-lock can be *downgraded* to a read lock
  (potentially allows greater concurrency)

Granularity of locks

- [redacted]
- [redacted]

2 Phase Locking

- most common locking protocol is known as *two-phase locking* (2PL)
- transactions operate in 2 distinct phases:
  - a *growing phase* during which the transaction acquires locks and a
  - *shrinking phase* during which it releases those locks

Rules for transactions obeying 2PL

1. transactions are well-formed, thus a transaction must acquire a lock on a data object *before* operating on it and all locks held by a transaction must be released when the transaction is finished
2. compatibility rules for locking are observed, thus no conflicting locks are held (write-write and read-write conflicts are forbidden)
3. once the transaction has released a lock, no new locks are acquired
4. all write locks are released together when the transaction commits (to ensure atomicity).

Rules for transactions obeying 2PL (cont)

Note:

- upgrading and downgrading of locks are possible under 2PL, with the restriction that downgrading is only permitted during the shrinking phase.
- it can be proved that the schedules produced by transactions which obey 2PL are guaranteed to be serialisable.

Deadlock

- Under 2PL, if a transaction T_1 requests a write-lock on data item x_i, which is currently locked by another transaction, T_2, there are two possibilities:

  Approach 1:

  Place T_1 on a queue for x_i awaiting release of lock by T_2

  Approach 2:

  Abort and rollback T_1

Deadlock (cont)

- with Approach 1, T_1 retains all the locks it currently holds and just enters a wait state
- Approach 1 can lead to **deadlock**

deadlock occurs when one transaction is waiting for a lock to be released by a second transaction, which is in turn waiting for a lock currently held by the first transaction

- Therefore, in approach 1, a **deadlock detection protocol** is needed

Deadlock (cont 3)

- With Approach 2, T_1 must release all its locks and restart (this could involve high overhead)
- Approach 2 is referred to as a **deadlock prevention protocol** because transaction releases all its locks on becoming blocked, so deadlock cannot occur

LiveLock

- transactions can be repeatedly rolled back or left in a wait state indefinitely, unable to acquire their locks, even though the system is not deadlocked => *livelock*, e.g. summary transactions
- priority system is needed to avoid livelock, whereby the longer a transaction has to wait, the higher its priority.

Deadlock Detection

- deadlock is generally detected by means of *wait-for* graphs.
- transactions are represented by nodes and blocked requests for locks are represented by labelled, directed edges (see Figure 6.11)
- deadlock is represented in the graph by the existence *cycle* between T_1 and T_2
- informally, a cycle in a graph forms a closed loop by which it is possible to start at one node, traverse the edges of the graph according to the directions of the arrows and get back to where you started

Figure 6.11

DeadLock Detection (cont)

- wait-for graph, G, can also be represented symbolically as:

$$G = T_1 \text{ ® } T_2 \text{ ® } T_1$$

- where T_1 ® T_2 indicates that transaction T_1 is waiting for transaction T_2 ; the ® represents the wait-for relationship)
- deadlock can occur between two transactions indirectly via a chain of intermediate transactions as shown in Figure 6.12

Figure 6.12

Deadlock Detection Graph

$G' = T_1$ ® T_2 ® T_3 ® T_4 ® T_5 ® T_1

- once detected, deadlock must be resolved by pre-emptying, i.e. aborting and rolling back, one of the transactions (any one will do since this will always break the cycle)

Deadlock Prevention

- how does the transaction manager decide whether or not to allow a transaction T_1, which has requested a lock on data item x_i currently held by transaction T_2, to wait and to *guarantee* that this waiting *cannot* give rise to deadlock?
- can force locks to be acquired in a certain data-dependent order (difficult to implement)
- alternatively, transactions can be ordered and ensure that all conflicting operations are executed in sequence according to this order;

Deadlock Prevention (Cont)

- deadlock is prevented by only allowing blocked transactions to wait under certain circumstances which will maintain this ordering
- ordering mechanism is generally based on **timestamps** (unique identifier assigned to transaction when it is launched)
- can ensure that *either* older transactions wait for younger ones (Wait-die) *or* vice versa (Wound-wait)
- figure 6.13 gives the algorithm for wait-die, while figure 6.14 gives the algorithm for wound-wait.

Wait Die - Fig 6.13

```
BEGIN
T1 requests lock on data item currently held by T2
if T1 is older than T2 (I.e. ts(T1)<ts(T2) )
   then T1 waits for T2 to commit or rollback
   else T1 is rolled back
endif
END
```

Wound Wait Fig 6.14

BEGIN
T1 requests lock on data item currently held by T2
if T1 is older than T2 (I.e. ts(T1)<ts(T2))
 then T2 is rolled back
 else T1 waits for T2 to commit or rollback
endif
END

Deadlock Prevention (Cont)

- if a transaction is rolled back, it retains its original timestamp (otherwise it could be repeatedly rolled back)
- wait-die and wound-wait use locks as the primary concurrency control mechanism and are therefore classified as *lock-based* rather than timestamp

The deadlock spectrum

- at one end of this spectrum, techniques are *deadlock-free*, i.e. deadlocks can never occur (require no run-time support)
- at the other end, techniques *detect and recover* from deadlock
- deadlock *prevention* methods lie in the middle,

TIMESTAMP METHODS

- no locks are involved and there can therefore be no deadlock.
- no waiting; transactions involved in conflict are simply rolled back and restarted.
- transactions are ordered globally in such a way that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict
- read or write only be allowed if the last update on that data item was carried out by an older transaction; otherwise the requesting transaction is restarted *and given a new timestamp*.

Timestamps (cont)

- **Thus:**

  Timestamp methods produce serialisable schedules, which are equivalent to the serial schedule defined by the timestamps of successfully committed transactions.
- each transaction is assigned a unique timestamp, when it is launched
- timestamps can use the system clock, a global counter or sequence number generator

Atomicity of transactions with timestamps

- with timestamp protocols we must prevent other transactions from seeing partial updates since there are no locks
- done by using *pre-writes (deferred update)*
- updates of uncommitted transactions are not written out to the database, but instead are written to a set of buffers, which are only flushed out to the DB when the transaction commits.
- has the advantage that when a transaction is aborted and restarted, no physical changes need to be made to the DB

Basic timestamping

- To implement basic timestamping, the following variables are required:

for each data item x

ts(read x) = the timestamp of the transaction which last read data item x

and

ts(write x) = the timestamp of the transaction which last updated data item x

and for each transaction T_i

ts(T_i) = the timestamp assigned to transaction T_i when it is launched

- transactions actually issue pre-writes to buffers rather than writes to the DB
- physical writes to the DB are only performed at commit

Pre-Write Algorithm Fig 6.18

BEGIN

Ti attempts to pre-write data item x

if x has been read or written to by a younger transaction (I.e. ts(Ti)<ts(read x) or ts(Ti)<ts(write x)

then reject Ti and restart as Tj

else accept pre-write: buffer (pre) write together with ts(Ti)

end if

END

Write Algorithm Fig 6.19

BEGIN

Ti attempts to update (I.e. write) data item x

if there is an update pending on x by an older transaction Tj (I.e. ts(Tj) < ts(Ti)

then Ti waits until Tj is committed or restarted

else Ti committs update and sets ts(write x) = ts(Ti)

end if

END

Read Operation using Basic Timestamping Fig 6.20

BEGIN

Ti attempts a read operation on data item x

if x has been updated by a younger transaction (I.e. ts(Ti)<ts(write x)

then reject read operation and restart Ti

else if there is an update pending on x by an older transaction Tj (I.e. ts(Tj)<ts(Ti))

then Ti waits for Tj to commit or restart

else accept read operation and set ts(read x) = max(ts(read x), ts(Ti))

endif

enf if

end

Timestamping (cont)

- Figures 6.18, 6.19 and 6.20 illustrate the algorithms for the pre-write, write and read operations, respectively, under basic timestamping
- Timestamping method is equivalent to applying exclusive locks on the data items between the pre-write and write operations
- Since all waits consist of younger transactions waiting for older transactions "deadlock" is not possible.