Database concurrency

Database users access shared data assumed to satisfy certain integrity constraints.

- Airline reservation system
  - 100’s of accessing terminals
  - only one agent can sell the last ticket for each flight
- Banking systems
- Car rental agency
- University registrar

101

Transactions

A transaction is an execution of a program having the following properties:

a) Atomicity

A transaction either happens or doesn’t. A transaction either completes and the results become visible or no results are visible.

b) Durability

The results of a completed transaction are permanently installed within D.B.

c) Consistency

Transactions preserve correctness of database.

102

A transaction has exactly one of two possible outcomes:

a) Commit

Program execution completes and the results become permanent in database.

b) Abort

Program execution was not successful. "Results" are not installed into database.

- Transactions may abort due to hardware failure, system error, bad input data, or as a means to ensure consistency.

103

Simple transaction model

Data entities

A, B, C, …

can be datapages, tuples, relations, files, etc.

Transactions built with

begin	creates private workspace
read(x)	look for x in workspace
	if not there, read x from secondary storage
write(x)	write x to secondary storage
end	end of transaction

104

These commands cannot provide transactions. There are interference problems.

P begin
read(A)   "increment" program
A = A + 1
write(A)
end

Consider transactions T1 and T2.

Time	T1	T2	A value
0	begin		7
1		begin	7
2		read(A)	7
3	read(A)		7
4	A = A+1		7
5		A = A+1	7
6		write(A)	8
7	write(A)		8
8	end		8
9		end	8

The A value has increased by one. Yet the "increment" transaction has been run twice.

Is this correct? Probably not.

We augment the commands available within transactions to avoid this type of interference problem.

Locks

- Each data entity has an associated lock.
- A transaction must possess the lock for data entity A before read(A) or write(A) commands are executed.
- At most one transaction can possess the lock to a data entity at any time.
- A transaction will release the lock to a data entity (unlock it) when it is not needed.

New commands

lock(-) synchronous

Returns when lock is available for transaction.

unlock(-)

Releases lock.

```
P       begin           "increment"
        lock A
        read A
        A = A+1
        write A
        unlock A
        end
```

Time	T1	T2	D.B. A value
0	begin		7
1		begin	7
2		lock A	7
3	lock A		7
4			7
5		read A	7
6		A = A+1	7
7		write A	8
8		unlock A	8
9		end	8
10			8
11	read A		8
12	A = A+1		8
13	write A		9
14	unlock A		9
15	end		9

Difficulties with respect to locks

fairness

deadlock For now we assume this can't happen. (more later)

109

Maintaining consistency

Even with locks there can be problems. We want a general protocol to ensure consistency.

We consider static case first.

Fixed set of transactions $T_1, T_2, \ldots, T_k$

Then we generalize to the dynamic (interesting) case in which transactions are not known in advance and yet consistency is maintained.

110

Notation set of transactions $T_1, T_2, T_3, \dots, T_k$.

- A schedule S for a set of transactions is a collated sequence of elementary steps from the transactions.
  a) We can identify the transaction T_i for each step in S
  b) The steps for each transaction T_i are in order.

  Example: T1 = abcd T2 = xyz

  A possible schedule for T1 and T2 is axybzcd

The schedule is the execution order for the CPU.

Non interference is assured if transactions run one-at-a-time.

- A schedule is serial if all steps of each transaction occur consecutively (so there is no interleaving)

We need greater concurrency.

Noninterference is assured if the effects of running the transactions concurrently is equivalent (same) as if they had run one-at-a-time in some order.

- A schedule is <u>serializable</u> if it is equivalent to a serial schedule.

Thisensures consistency, assuming each transaction is consistent.

Other issues:

determinacy

fairness

<u>We elect only to insure consistency of database.</u>

Given a set of transactions we don't require (or care) that the equivalent serial order always be the same. Fairness and determinacy are not insured.

Simplest transactions

- Only commands lock and unlock.

A transaction <u>locks</u> and <u>unlocks</u> any data entity at most one time.

The following discussions utilizing this model present the essential points (for more interesting transaction models)

Equivalence [syntactic version only]

Consider only two commands: lock and unlock.

- For each transaction T and each pair, associate a

  lock A
  …
  unlock A

unique uninterpreted function f^{T}_{A}.

For schedule S, the resulting value for data entity A is a formula

$f^{Ti1}_{A}\ f^{Ti2}_{A} \ldots f^{Tik}_{A}(A)$

S

T_1 lock A
…
T_2 lock B
T_1 unlock A
T_2 unlock B
…
T_2 lock A
T_2 unlock A
…
T_1 lock B
T_1 unlock B

f^{T1}_{A}
f^{T2}_{B}
f^{T2}_{A}
f^{T1}_{B}

Final value
A: $f^{T2}_{A}\ f^{T1}_{A}(A)$
B: $f^{T1}_{B}\ f^{T2}_{B}(B)$

- Two schedules S_1 and S_2 are equivalent if the formula for each data entity is the same for both schedules.

Example 1. Schedule and values.

T1	lock B	f^{T1}_B	T2	lock A	f^{T2}_A
	unlock B			unlock A	
T3	lock A	f^{T3}_A		lock B	f^{T2}_B
	unlock A			unlock B	

OR

S1	T2	lock A	T2	lock A	S2
	T2	unlock A	T2	unlock A	
	T3	lock A	T1	lock B	
	T3	unlock A	T1	unlock B	
	T1	lock B	T3	lock A	
	T1	unlock B	T3	unlock A	
	T2	lock B	T2	lock B	
	T2	unlock B	T2	unlock B	

A: $f^{T3}_A\, f^{T2}_A(A)$

B: $f^{T2}_B\, f^{T1}_B(B)$

S1 and S2 are equivalent

- In fact, S1 and S2 have a serial equivalent T1, T2, T3

Example 2. Schedule and values.

T1	lock B	f^{T1}_A	T2	lock B	f^{T2}_B
	lock B			unlock B	
	unlock A	f^{T1}_B		lock A	f^{T2}_A
	unlock B			unlock A	

S	T1	lock A	
	T2	lock B	$f^{T2}_A\, f^{T1}_A(A)$
	T2	unlock B	$f^{T1}_B\, f^{T2}_B(B)$
	T1	lock B	
	T1	unlock A	
	T2	lock A	
	T2	unlock A	
	T1	unlock B	

T1 comes before T2 wrt A

T2 comes before T1 wrt B

S is not serializable.

Serializability test (simplest transactions)

Input: schedule S

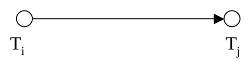
Output: yes   if serializable

no   otherwise

Step 1. Create a directed graph Gs.

[nodes]   correspond to transactions

[arcs]



T_i: unlock A in S

locate next following lock A statement in S

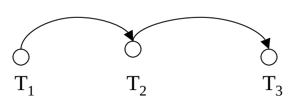
T_j: lock A in S

Step 2. Topologically sort Gs if possible (i.e if G is acyclic)

Return yes   if possible

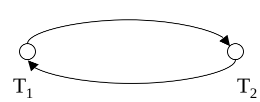
no   otherwise

Previous example 1 revisited



G_S

Previous example 2



G_S

- Comment regarding algorithm

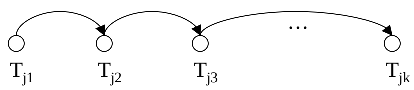
  If topological sort IS possible, the sort provides an equivalent serial schedule.

Algorithm is correct

Suppose G_S has no cycles. Let ♦ be the resulting serial schedule.

For each data entity A, consider the sequence of transactions locking the entity in S.

$T_{j1}, T_{j2}, \dots, T_{jk}$

Arcs 

must exist in G_S.

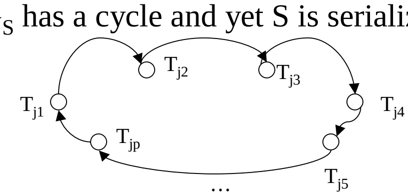
∴ Final value of A is

$f^{Tjk}_{A}\, f^{Tjk-1}_{A} \dots f^{T2}_{A}\, f^{Tj1}_{A}(A)$

In either S or ♦. So S and ♦ must be equivalent.

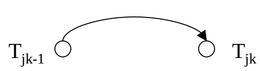
Algorithm is complete

Assume G_S has a cycle and yet S is serializable ≡ ♦.



In G_S

Suppose T_{jk} is first transaction from cycle in ♦. And



Exists in G_S because of data object A.

Final value of A must differ for S and ♦.

Here's why

In S

… f^{Tjk}_{A} f^{Tjk-1}_{A} … (A)

In 🌢

… f^{Tjk}_{A}(A)

So S and 🌢 cannot be equivalent, contradicting our hypothesis.

∴ G_S having a cycle ⇒ S is not serializable.

☒

123

Dynamic version ☺

- 2-phase locking protocol

  A transaction T satisfies the 2-phase locking protocol if all lock commands of T precede the unlock commands of T. ☒

Previous example 1.

T1, T3	Trivially 2-phase
T2	Not 2-phase.

Example 2.

T1	2-phase
T2	Not 2-phase

124

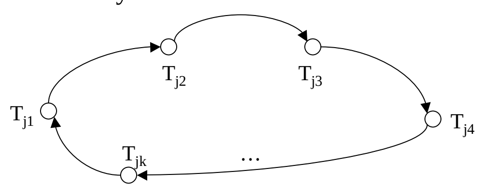
Result

Any schedule S in which each transaction satisfies the 2-phase locking protocol is serializable.

Proof:

Assume S contains only transactions satisfying the 2-phase locking protocol and yet it is not serializable.

G_S must contain a cycle.



125

Suppose T_{j1} is the first transaction in cycle to lock something in S.

Tracing around the cycle we have

T_{j1} locks and unlocks something
T_{j2} locks and unlocks something
…
T_{jk} locks and unlocks something
T_{j1} locks and unlocks something else

So T_{j1} contains a lock command after an unlock command and cannot satisfy the 2-phase lock protocol, contradicting our hypothesis.

∴ The result follows.

126

This result is best possible

For every non 2-phase locked transaction T1 there exists a 2-phase locked transaction T2 and a schedule S such that S is not serializable.

Basic idea:

T1	lock A	T2	lock A
	unlock A		lock B
	lock B		unlock A
	unlock B		unlock B

S T1 lock A, T1 unlock A,
T2 …………………..
T1 lock B, T1 unlock B.

S is not serializable.

Improved concurrency

Two locking modes

shared locks	"read only"
exclusive locks	"read/write"

- Any number of transactions can hold a share(A) lock at any time.
- At most one transaction can hold the exclusive(A) lock at any time. No transaction can hold a share(A) lock at the same time.

Commands

exclusive(-), share(-), unlock(-)

Equivalence of schedules

Schedules S1 and S2 are equivalent provided

a) S1 and S2 produce same values for each data entity

AND

b) shared locks are applied at times when data entity has same value in both schedules.

Syntactic equivalence

- $\underbrace{\text{exclusive(A) … unlock(A)}}_{f^T_A}$
- nothing for share(-)

Test for serializability

Input: schedule S

Output: yes if serializable

no otherwise

Step 1. Construct graph G_S

[nodes]: one per transaction

[arcs]: three cases.

i) $T_i \longrightarrow T_j$

T_i: exclusive lock of A

T_j: next exclusive lock of A in S.

Comment: these arcs ensure data items are written in the same order in S and the derived (serial) schedule.

ii) $T_i \rightarrow T_j$

T_i : share(A)
T_j : <u>next</u> exclusive(A) <u>in S</u>

iii) $T_i \rightarrow T_j$

T_i : exclusive(A)
T_j : <u>any</u> share(A) in S before next exclusive(A)

Comment. Cases ii and iii ensure point b of equivalence

Step 2. Topologically sort G_S if possible (graph is acyclic)

Return yes and order if possible

no otherwise. ⊠

131

Serializability test is correct

✓ Assume G_S has a cycle. Then S has no serial equivalent.

✓ Assume G_S has no cycle. Let 🌢 be the derived serial schedule:

--Final values in S and 🌢 are identical.

--Share locks must see same values in S and 🌢 by construction. The command

$T_{i:}$ share(A)

must have identical sequences of exclusive(A) commands preceding it in both S and 🌢.

∴ S ≡ 🌢.

Completeness is similar. ⊠

132