IN3020&4020 – Database Systems (2026)
Part 2: Transaction management
Lectures 19, 20, 21.1: Concurrency control techniques

23, 25 March, 8 April

Egor V. Kostylev
IFI, University of Oslo
egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases ✓

Part 2. Transaction management in relational databases

- Transaction processing concepts (3 lectures) ✓
- Concurrency control techniques (3 lectures)
- Database recovery techniques (2 lectures)

Part 3. NoSQL DBMS

Materials to read about transaction management

Part 2. Transaction management in relational databases

Fundamentals of Database Systems by R. Elmasri and S.B. Navathe (7th edition): Part 9 (close to my presentation)

- Transaction processing concepts (3 lectures)

  Chapter 20

- Concurrency control techniques (3 lectures)

  Chapter 21

- Database recovery techniques (2 lectures)

  Chapter 22

Materials to read about transaction management

Part 2. Transaction management in relational databases

Fundamentals of Database Systems by R. Elmasri and S.B. Navathe (7th edition): Part 9 (close to my presentation)

- Transaction processing concepts (3 lectures)
  Chapter 20
- Concurrency control techniques (3 lectures)
  Chapter 21
- Database recovery techniques (2 lectures)
  Chapter 22

Database Systems: the Complete Book by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Parts of Section 18

etc.

Reminders – 1

Transaction is a sequence of operations (not arbitrary)

- `begin`, `end`, `commit`,
  `abort`, `read`(X), `write`(X), variable updates, etc.
- we use abbreviations (`b`, `e`, `c`, `a`, `r`(X), `w`(X))
- we may omit operations (e.g., use only `r`(X) and `w`(X))
- we often append transaction ids (e.g., $\texttt{b}_1$, $\texttt{r}_1(X)$, $\texttt{r}_2(X)$, $\texttt{w}_3(X)$)
- transactions read from and write to the common disk,
  but have their own main memory

Can be committed, aborted or partial

Reminders – 1

Transaction is a sequence of operations (not arbitrary)

- `begin`, `end`, `commit`, `abort`, `read`(X), `write`(X), variable updates, etc.
- we use abbreviations (`b`, `e`, `c`, `a`, `r`(X), `w`(X))
- we may omit operations (e.g., use only `r`(X) and `w`(X))
- we often append transaction ids (e.g., $\texttt{b}_1$, $\texttt{r}_1(X)$, $\texttt{r}_2(X)$, $\texttt{w}_3(X)$)
- transactions read from and write to the common disk, but have their own main memory

Can be committed, aborted or partial

Schedule (history, or execution plan) S of transactions $T_1, \ldots, T_n$ is a (total) ordering of the operations of $T_1, \ldots, T_n$

- operations of different transitions can interleave
- operations of each T_i are in the same order as in T_i

Complete schedule: all $T_1, \ldots, T_n$ are committed or aborted

Reminders – 2

ACID principles:

- Atomicity: Transaction performed in its entirety or not at all
- Consistency: The database should always remain consistent
- Isolation: Transaction should not interfere with others
- Durability: Changes of committed transactions must persist

Reminders – 2

ACID principles:

- Atomicity: Transaction performed in its entirety or not at all
- Consistency: The database should always remain consistent
- Isolation: Transaction should not interfere with others
- Durability: Changes of committed transactions must persist

We agreed to ensure isolation by enforcing (conflict-)serialisability:

- schedule is serial if no transaction begins when there is another active (non-committed and non-aborted) transaction
- schedule is (conflict-)serialisable if it is (conflict-)equivalent to a serial one
- two schedules (of the same transactions) are conflict-equivalent if the relative order of every conflict is the same in both
- a conflict is a pair of operations from different transactions at least one of which is `write`

Ensuring serialisability in DBMSs

Naive approach:

- run all transactions in an arbitrary way
- construct the precedence graph in the process
- when done, check if the graph is acyclic
- abort and restart everything if not

Obvious drawbacks

Ensuring serialisability in DBMSs

Naive approach:

- run all transactions in an arbitrary way
- construct the precedence graph in the process
- when done, check if the graph is acyclic
- abort and restart everything if not

Obvious drawbacks

Instead, DBMSs arrange transactions
(individually and in the schedule)
using concurrency control protocols (i.e., set of rules)
that ensure serialisation by construction

In this set of lectures, we consider such protocols

Lectures Plan

1. Serialisability via two-phase locking

1.1. Types of locks

1.2. Two-phase locking protocol

1.3. Deadlocks

2. Serialisability via timestamps

3. Multiversioning for concurrency control

4. Concurrency with multiple granularity

5. Other issues and conclusion

1. Two-phase locking

1.1. Types of locks

Locks

A lock: a variable associated with a data item

- unique LKD(X) for each data item X
- describes status for operations that can be applied

Locks

A lock: a variable associated with a data item

- unique LKD(X) for each data item X
- describes status for operations that can be applied

We will consider

- binary locks:
  LKD(X) takes values locked or unlocked (or 1 and 0)
- shared/exclusive locks:
  LKD(X) takes values read-locked, write-locked, or unlocked

These variables are managed by
concurrency control subsystem (transaction manager)

Binary locks

Binary locks:

for each item X, variable LKD(X) can have one of the two values:

- LKD(X) is locked (or 1) means that X is locked by a transaction and cannot be accessed by any other transaction
- LKD(X) is unlocked (or 0) means that X is available and can be accessed when requested
- schedules start with all unlocked

Binary locks in transactions

Transaction with binary locks: can use two operations `lock(X)` and `unlock(X)` following several rules:

- all `read(X)` and `write(X)` must be between the two
- is well-formed: no unlocking without locking, etc.

These operations are implemented such that

- when `lock(X)` is issued
  - if LKD(X) = 1, then the transaction is forced to wait for LKD(X) = 0 (in the schedule)
  - if LKD(X) = 0, it is set to 1 and the transaction can proceed
- when `unlock(X)` is issued, LKD(X) is set to 0

run by transaction manager (concurrency subsystem), see below

Binary locks in transactions

Transaction with binary locks: can use two operations `lock(X)` and `unlock(X)` following several rules:

- all `read(X)` and `write(X)` must be between the two
- is well-formed: no unlocking without locking, etc.

These operations are implemented such that

- when `lock(X)` is issued
  - if LKD(X) = 1, then the transaction is forced to wait for LKD(X) = 0 (in the schedule)
  - if LKD(X) = 0, it is set to 1 and the transaction can proceed
- when `unlock(X)` is issued, LKD(X) is set to 0

run by transaction manager (concurrency subsystem), see below

As a result, certain schedules are disallowed

Example: Transactions with binary locks

T_1
$\texttt{lock}(X);$
$\texttt{read}(X);$
$X := X - 1;$
$\texttt{write}(X);$
$\texttt{unlock}(X);$
$\texttt{lock}(Y);$
$\texttt{read}(Y);$
$Y := Y + 1;$
$\texttt{write}(Y);$
$\texttt{unlock}(Y);$

T_2
$\texttt{lock}(X);$
$\texttt{lock}(Y);$
$\texttt{read}(X);$
$\texttt{unlock}(X);$
$X := X + 1;$
$\texttt{lock}(X);$
$\texttt{write}(X);$
$\texttt{unlock}(Y);$
$\texttt{unlock}(X);$

$T_1 : \texttt{l}_1(X), \texttt{r}_1(X), \texttt{w}_1(X), \texttt{u}_1(X), \texttt{l}_1(Y), \texttt{r}_1(Y), \texttt{w}_1(Y), \texttt{u}_1(Y)$: Ok!

$T_2 : \texttt{l}_2(X), \texttt{l}_2(Y), \texttt{r}_2(X), \texttt{u}_2(X), \texttt{l}_2(X), \texttt{w}_2(X), \texttt{u}_2(Y), \texttt{u}_2(Y)$: Ok!

Note: T_2 locks and unlocks X twice and locks Y without a need

Example: Transactions with binary locks 2

T_1
$\texttt{lock}(X);$
$\texttt{read}(X);$
$X := X - 1;$
$\texttt{write}(X);$
$\texttt{unlock}(X);$
$\texttt{read}(Y);$
$\texttt{lock}(Y);$
$Y := Y + 1;$
$\texttt{write}(Y);$
$\texttt{unlock}(Y);$

T_2
$\texttt{lock}(X);$
$\texttt{lock}(Y);$
$\texttt{read}(X);$
$\texttt{unlock}(X);$
$X := X + 1;$
$\texttt{lock}(X);$
$\texttt{write}(X);$
$\texttt{unlock}(X);$

$T_1 : \texttt{l}_1(X), \texttt{r}_1(X), \texttt{w}_1(X), \texttt{u}_1(X), \texttt{r}_1(Y), \texttt{l}_1(Y), \texttt{w}_1(Y), \texttt{u}_1(Y)$: Not ok

$T_2 : \texttt{l}_2(X), \texttt{l}_2(Y), \texttt{r}_2(X), \texttt{u}_2(X), \texttt{l}_2(X), \texttt{w}_2(X), \quad \texttt{u}_2(Y)$: Not ok

Note: T_1 reads Y without locking, and T_2 does not unlock Y

Example: Schedules with binary locks

$S_1 : \mathtt{l}_1(X), \mathtt{r}_1(X), \mathtt{u}_1(X), \quad \mathtt{l}_2(X), \mathtt{r}_2(X), \mathtt{u}_2(X), \quad \mathtt{l}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Ok!

$S_2 : \mathtt{l}_1(X), \mathtt{r}_1(X), \quad \mathtt{l}_2(X), \mathtt{u}_1(X), \quad \mathtt{r}_2(X), \mathtt{u}_2(X), \quad \mathtt{l}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Ok!

$S_3 : \mathtt{l}_1(X), \mathtt{r}_1(X), \quad \mathtt{l}_2(X), \mathtt{r}_2(X), \mathtt{u}_1(X), \quad \mathtt{u}_2(X), \quad \mathtt{l}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Not ok (implementation of $\mathtt{l}_2(X)$ is 'broken': should not allow to continue T_2 with $\mathtt{r}_2(X)$ before $\mathtt{u}_1(X)$)

Note: locks (i.e., the rules and implementations as above)
do not guarantee serialisability by themselves;
we will see how they can be used to ensure serialisability

Lock table

Concurrency control subsystem of a DBMS has a lock manager module that relies on a lock table:

- has schema [Data_item_name, Locking_transaction]
- keeps only locked items (others are unlocked)

Lock manager keeps track of and controls access to locks
by checking the rules for transactions and
enforcing allowed schedules (waiting, etc.)

Lock table

Concurrency control subsystem of a DBMS has a lock manager module that relies on a lock table:

- has schema [Data_item_name, Locking_transaction]
- keeps only locked items (others are unlocked)

Lock manager keeps track of and controls access to locks
by checking the rules for transactions and
enforcing allowed schedules (waiting, etc.)

Binary locking is too restrictive anyway
so before moving to the description of how they are used,
we consider more advanced locks

Shared/exclusive locks

Shared/exclusive (or read/write) locks idea: we can allow sharing for several reading transactions (but still exclusive writes)

Shared/exclusive locks

Shared/exclusive (or read/write) locks idea: we can allow sharing for several reading transactions (but still exclusive writes)

Shared/exclusive (or read/write) locks formally: for each item X

- LKD(X) is read-locked (or share-locked) means that X is read by a transaction
- auxiliary numeric variable #READS(X) is maintained, which keeps the number of reading transactions
- LKD(X) is write-locked (or exclusive-locked) means that X is exclusively locked for writing by some transaction and cannot be accessed by others
- LKD(X) is unlocked means that X is available for access (with locking)
- schedules start with all LKD(X) = unlocked and #READS(X) = 0

Shared/exclusive locks

Shared/exclusive (or read/write) locks idea: we can allow sharing for several reading transactions (but still exclusive writes)

Shared/exclusive (or read/write) locks formally: for each item X

- LKD(X) is read-locked (or share-locked) means that X is read by a transaction
- auxiliary numeric variable #READS(X) is maintained, which keeps the number of reading transactions
- LKD(X) is write-locked (or exclusive-locked) means that X is exclusively locked for writing by some transaction and cannot be accessed by others
- LKD(X) is unlocked means that X is available for access (with locking)
- schedules start with all LKD(X) = unlocked and #READS(X) = 0

Lock table stores this information appropriately (maintained by lock manager)

Shared/exclusive locks transaction operations

Transaction with shared/exclusive locks: can use three operations `read_lock(X)`, `write_lock(X)` and `unlock(X)` with rules:

- each `read(X)` is after `read_lock(X)` or `write_lock(X)` (with no `unlock(X)` in between)
- each `write(X)` is after `write_lock(X)` (with no `unlock(X)` in between)
- each `read_lock(X)` and `write_lock(X)` has `unlock(X)` at some point after

Shared/exclusive locks transaction operations

Transaction with shared/exclusive locks: can use three operations `read_lock(X)`, `write_lock(X)` and `unlock(X)` with rules:

- each `read(X)` is after `read_lock(X)` or `write_lock(X)` (with no `unlock(X)` in between)
- each `write(X)` is after `write_lock(X)` (with no `unlock(X)` in between)
- each `read_lock(X)` and `write_lock(X)` has `unlock(X)` at some point after

- no changing of lock type before unlocking;
  that is, no `read_lock(X)` after `write_lock(X)` without `unlock(X)` between

  (this may be changed, see lock conversion below)

Implementation of read_lock

The routine that the transaction manager runs when it receives read_lock(X) from a transaction:

```
read_lock(X):
B:   if   LKD(X) = "unlocked"
          then begin   LKD(X) ← "read-locked";
                       #READS(X) ← 1
               end
     else if   LKD(X) = "read-locked"
          then    #READS(X) ←    #READS(X) + 1
     else begin
               wait (until   LKD(X) = "unlocked"
                     and the lock manager wakes up the transaction);
               go to B
               end;
```

Implementation of write_lock

The routine that the transaction manager runs when it receives write_lock(X) from a transaction:

```
write_lock(X):
B:   if   LKD(X) = "unlocked"
          then   LKD(X) ← "write-locked"
     else begin
               wait (until   LKD(X) = "unlocked"
                    and the lock manager wakes up the transaction);
               go to B
               end;
```

Implementation of unlock

The routine that the transaction manager runs when it receives unlock(X) from a transaction:

```
unlock (X):
    if   LKD(X) = "write-locked"
         then begin   LKD(X) ← "unlocked";
                      wakeup one of the waiting transactions, if any
                      end
    else it   LKD(X) = "read-locked"
         then begin
                      #READS(X) ←    #READS(X) −1;
                      if   #READS(X) = 0
                           then begin   LKD(X) = "unlocked";
                                        wakeup one of the waiting transactions, if any
                                        end
                      end;
```

Examples: Schedules with shared/exclusive locks

$S_1 : \mathtt{rl}_1(X), \mathtt{r}_1(X), \mathtt{u}_1(X), \quad \mathtt{rl}_2(X), \mathtt{r}_2(X), \mathtt{u}_2(X), \quad \mathtt{wl}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Ok!

$S_1' : \mathtt{rl}_1(X), \mathtt{r}_1(X), \mathtt{u}_1(X), \quad \mathtt{rl}_2(X), \mathtt{r}_2(X), \mathtt{u}_2(X), \quad \mathtt{rl}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Not ok (write with read lock)

$S_1'' : \mathtt{wl}_1(X), \mathtt{r}_1(X), \mathtt{u}_1(X), \quad \mathtt{rl}_2(X), \mathtt{r}_2(X), \mathtt{u}_2(X), \quad \mathtt{wl}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Ok!

$S_3 : \mathtt{rl}_1(X), \mathtt{r}_1(X), \quad \mathtt{rl}_2(X), \mathtt{r}_2(X), \mathtt{u}_1(X), \quad \mathtt{u}_2(X), \quad \mathtt{wl}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Ok!

$S_3' : \mathtt{rl}_1(X), \mathtt{r}_1(X), \quad \mathtt{wl}_2(X), \mathtt{w}_2(X), \mathtt{u}_1(X), \quad \mathtt{u}_2(X), \quad \mathtt{wl}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Not ok (implementation of $\mathtt{wl}_2(X)$ is 'broken': should be exclusive)

$S_3'' : \mathtt{rl}_1(X), \mathtt{r}_1(X), \quad \mathtt{rl}_2(X), \mathtt{r}_2(X), \quad \mathtt{u}_2(X), \quad \mathtt{wl}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Not ok (T_1 is not following the rules: a second lock without unlocking)

Lock conversion

Transaction rule

- no changing of lock type before unlocking
  (e.g., no `read_lock(X)` after `write_lock(X)` without `unlock(X)` between)

can be dropped, thus allowing for lock conversion—that is, lock upgrade and downgrade

This may be obviously beneficial for concurrency

Implementations of `read_lock(X)`, `write_lock(X)`, and `unlock(X)` need modifications (left for homework)

Example: Shared/exclusive locks with conversions

All above are the same, but

$S_3'' : \mathtt{rl}_1(X), \mathtt{r}_1(X), \quad \mathtt{rl}_2(X), \mathtt{r}_2(X), \quad \mathtt{u}_2(X), \quad \mathtt{wl}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Now ok!

Example: Shared/exclusive locks with conversions

All above are the same, but

$S_3'' : \mathtt{rl}_1(X), \mathtt{r}_1(X), \quad \mathtt{rl}_2(X), \mathtt{r}_2(X), \quad \mathtt{u}_2(X), \quad \mathtt{wl}_1(X), \mathtt{w}_1(X), \mathtt{u}_1(X)$

Now ok!

Still, do not guarantee serialisability by themselves,

but are needed to guarantee it in the protocols below

1.2. Two-phase locking protocol

Two-phase locking: definition

Conversion-free transaction (binary or shared/exclusive) follows two-phase locking protocol if it follows the rule

- all locking operations are before all unlocking operations

Two phases:

- locking (first, expanding, or growing) phase
- unlocking (second or shrinking) phase

Two-phase locking: definition

Conversion-free transaction (binary or shared/exclusive) follows two-phase locking protocol if it follows the rule

- all locking operations are before all unlocking operations

Two phases:

- locking (first, expanding, or growing) phase
- unlocking (second or shrinking) phase

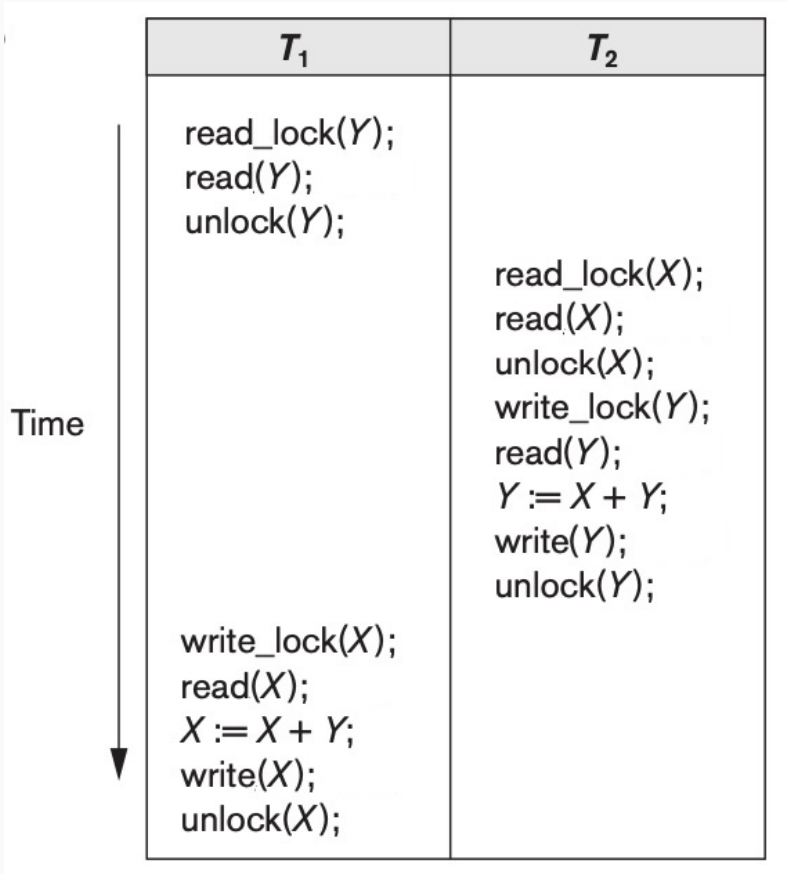
If conversion is allowed,

- all upgrades (from read- to write-lock) in growing phase
- all downgrades (from write- to read-lock) in shrinking phase

Two-phase locking: negative example

T_1	T_2

read_lock(Y);
read(Y);
unlock(Y);
write_lock(X);
read(X);
$X := X + Y$;
write(X);
unlock(X);
	read_lock(X);
	read(X);
	unlock(X);
	write_lock(Y);
	read(Y);
	$Y := X + Y$;
	write(Y);
	unlock(Y);

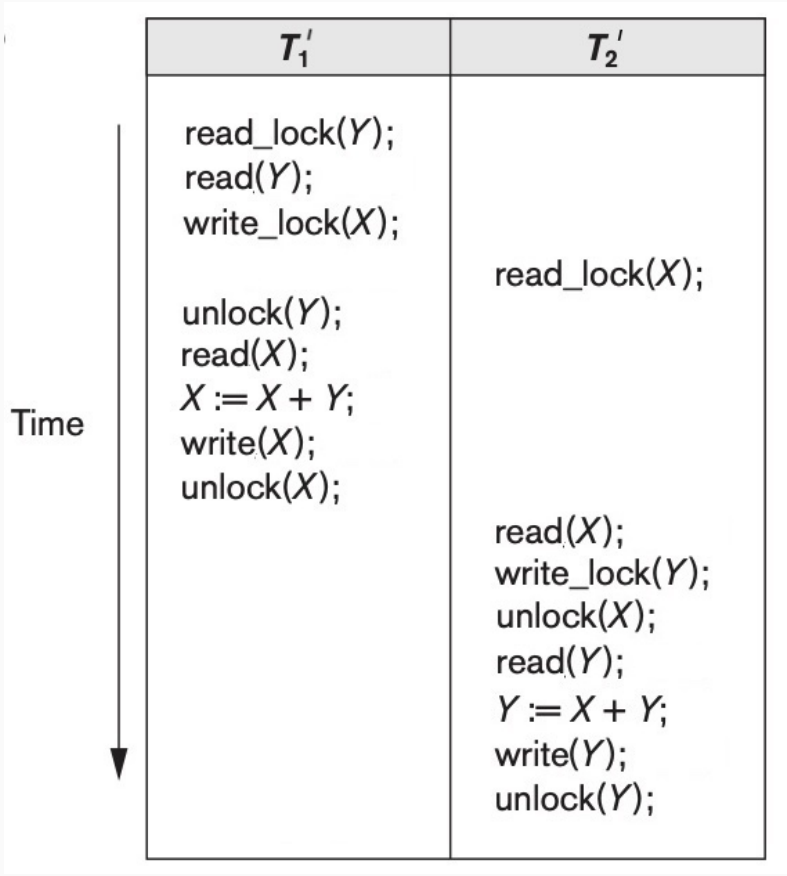


Transactions T_1 and T_2 (left) do not follow the two-phase locking protocol, so they allow for a non-serialisable schedule (right)

Two-phase locking: positive example

T_1'	T_2'

read_lock(Y);
read(Y);
write_lock(X);
unlock(Y);
read(X);
$X := X + Y$;
write(X);
unlock(X);
	read_lock(X);
	read(X);
	write_lock(Y);
	unlock(X);
	read(Y);
	$Y := X + Y$;
	write(Y);
	unlock(Y);



Versions T_1' and T_2' (left) follow the two-phase locking protocol, so no non-serialisable schedule complies the rules

Serialisability by two-phase locking

Two-phase locking guarantees serialisability
(i.e., every allowed schedule of transactions following the rules of the two-phase locking protocol is (conflict-)serialisable)

As the result, we can create a serialisable schedule operation by operation, without knowing the transactions in full

Serialisability by two-phase locking

Two-phase locking guarantees serialisability
(i.e., every allowed schedule of transactions following the rules of the two-phase locking protocol is (conflict-)serialisable)

As the result, we can create a serialisable schedule operation by operation, without knowing the transactions in full

Drawback 1: Some serialisable schedules do not comply the protocol, thus limiting the amount of concurrency

(formally, there is a conflict-serialisable schedule without locks that cannot be converted to an allowed schedule of transactions following the protocol by inserting locking and unlocking operations)

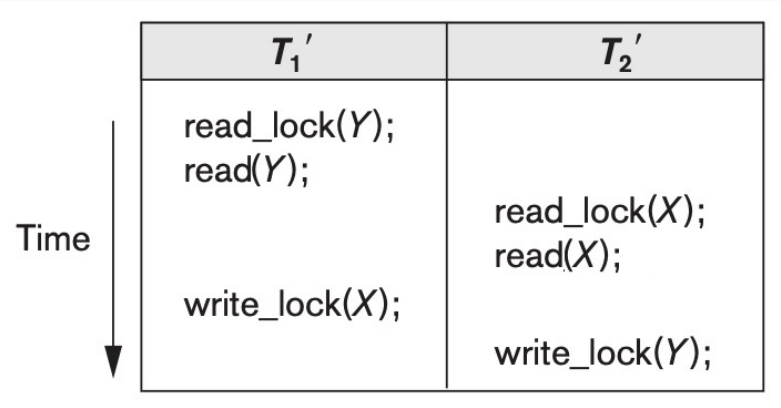
Homework: come up to an example

Serialisability by two-phase locking

Two-phase locking guarantees serialisability
(i.e., every allowed schedule of transactions following the rules of the two-phase locking protocol is (conflict-)serialisable)

As the result, we can create a serialisable schedule operation by operation, without knowing the transactions in full

Drawback 2: we may end up in a deadlock

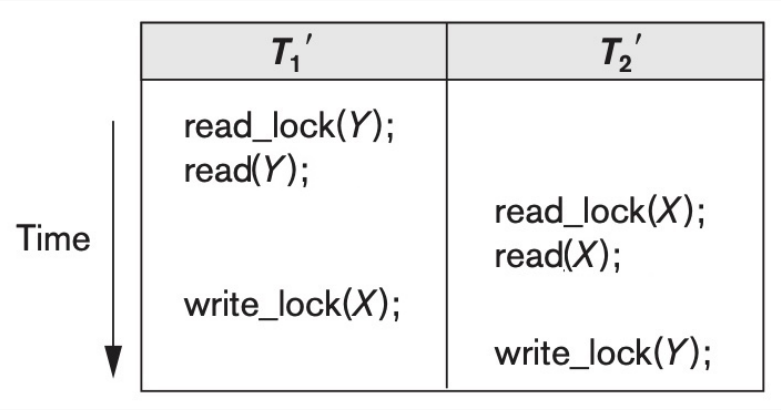


Serialisability by two-phase locking

Two-phase locking guarantees serialisability
(i.e., every allowed schedule of transactions following the rules of the two-phase locking protocol is (conflict-)serialisable)

As the result, we can create a serialisable schedule operation by operation, without knowing the transactions in full

Drawback 2: we may end up in a deadlock



We will look at methods to deal with deadlocks in few minutes

Variations of two-phase locking

More constrained versions of two-phase locking often guarantee better properties (while sacrificing even more concurrency):

- conservative (or static) two-phase locking:
  all locking of a transaction are at its beginning
  deadlock-free, requires to know the read and write sets at the beginning

Variations of two-phase locking

More constrained versions of two-phase locking often guarantee better properties (while sacrificing even more concurrency):

- conservative (or static) two-phase locking:

  all locking of a transaction are at its beginning

  deadlock-free, requires to know the read and write sets at the beginning

- strict two-phase locking:

  all write unlocking of a transaction are at its end

  not deadlock-free, used in practice, guarantees strict schedules

Variations of two-phase locking

More constrained versions of two-phase locking often guarantee better properties (while sacrificing even more concurrency):

- conservative (or static) two-phase locking:

  all locking of a transaction are at its beginning

  deadlock-free, requires to know the read and write sets at the beginning

- strict two-phase locking:

  all write unlocking of a transaction are at its end

  not deadlock-free, used in practice, guarantees strict schedules

- rigorous two-phase locking:

  all unlocking of a transaction are at its end

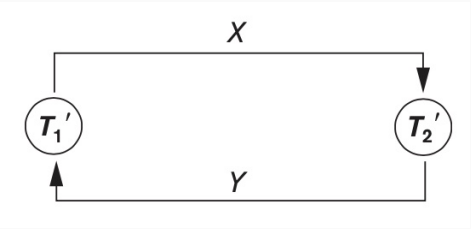
  not deadlock-free, easy to implement, guarantees strict schedules, very limited concurrency

1.3. Deadlocks

Deadlock problem

Two (or more) transactions may be stuck in a deadlock, where each transaction is waiting for another transaction, making a 'waiting cycle'

Time	T_1'	T_2'
↓	read_lock(Y);	
	read(Y);	
		read_lock(X);
		read(X);
	write_lock(X);	
		write_lock(Y);



This may happen for all types of locks,

so we do not assume a specific type below

Deadlock avoidance: Overview

We will consider several groups of (sub-)protocols
to deal with deadlocks:

- (naive)
- deadlock prevention
- deadlock detection
- timeouts

Deadlock avoidance: Overview

We will consider several groups of (sub-)protocols
to deal with deadlocks:

- (naive)
- deadlock prevention
- deadlock detection
- timeouts

Most of protocols may roll back (i.e., abort and restart)
some transactions

- cascadeless (i.e., recoverable) due to two-phase locking
(all deadlocked transactions in the growing phase,
so no one have read what they have written)
- may face and have to deal with starvation:
a transaction rolls back over and over again indefinitely

Deadlock avoidance protocols: naive approaches

We have seen one method:

- conservative two-phase locking (not practical)

Deadlock avoidance protocols: naive approaches

We have seen one method:

conservative two-phase locking (not practical)

Another simple method:

locking according to a predefined order of all items

- all items are ordered in advance (e.g., by address)
- all locking of a transaction is in the increasing order

Deadlock avoidance protocols: naive approaches

We have seen one method:

conservative two-phase locking (not practical)

Another simple method:

locking according to a predefined order of all items

- all items are ordered in advance (e.g., by address)
- all locking of a transaction is in the increasing order
- no deadlocks by construction
- not practical due to
  - limited concurrency
  - unreasonable restrictions on transactions

Deadlock prevention protocols: Overview

We will consider four more methods for deadlock prevention:

- timestamp-based (wait-die and wound-wait)
- waiting-based (no-waiting and cautious-waiting)

All four avoid deadlocks, but often force transaction aborts without a real deadlock

Deadlock prevention
is not a popular approach in practice due to high overhead,
but may be useful when transactions are long and use many items

Timestamp-based deadlock prevention: Wait-die rule

Transaction timestamp:

a unique number $TS(T)$ assigned to each transaction T so that $TS(T') < TS(T)$ for each T' started before (for the first time)

Timestamp-based deadlock prevention: Wait-die rule

Transaction timestamp:

a unique number $TS(T)$ assigned to each transaction T so that $TS(T') < TS(T)$ for each T' started before (for the first time)

Wait-die rule:

let transaction T try to lock an item that is already locked by T'

- if $TS(T) < TS(T')$ (i.e., T is older than T'), then T waits for T' to release
- otherwise (i.e., T is younger) T aborts (i.e., 'dies') and restarts with the same timestamp

The older proceeds (maybe, after waiting) and the younger restarts, so the wait-die rule guarantees no deadlocks, no starvation

Timestamp-based deadlock prevention: Wound-wait rule

Transaction timestamp:

a unique number $TS(T)$ assigned to each transaction T so that $TS(T') < TS(T)$ for each T' started before (for the first time)

Wound-wait rule:

let transaction T try to lock an item that is already locked by T'

- if $TS(T) < TS(T')$ (i.e., T is older than T'), then T' aborts (i.e., is 'wounded') and restarts with the same timestamp
- otherwise (i.e., T is younger) T waits for T' to release

The older proceeds and the younger restarts (maybe, after waiting), so the wound-wait rule guarantees no deadlocks, no starvation

Examples of wait-die and wound-wait

When T_1 tries to lock X that is already locked by T_2

- if T_1 is older
  - wait-die: T_1 waits

    $\mathtt{b}_1, \ldots, \mathtt{b}_2, \ldots, \mathtt{l}_2(X), \ldots, \mathtt{l}_1(X),\ [T_1\ \textit{waits}] \ldots, \mathtt{u}_2(X), \ldots$
  - wound-wait: T_2 dies

    $\mathtt{b}_1, \ldots, \mathtt{b}_2, \ldots, \mathtt{l}_2(X), \ldots, \mathtt{l}_1(X), \mathtt{a}_2, \ldots$

- if T_1 is younger
  - wait-die: T_1 dies

    $\mathtt{b}_2, \ldots, \mathtt{b}_1, \ldots, \mathtt{l}_2(X), \ldots, \mathtt{l}_1(X), \mathtt{a}_1, \ldots$
  - wound-wait: T_1 waits

    $\mathtt{b}_2, \ldots, \mathtt{b}_1, \ldots, \mathtt{l}_2(X), \ldots, \mathtt{l}_1(X),\ [T_1\ \textit{waits}] \ldots, \mathtt{u}_2(X), \ldots$

Waiting-based deadlock prevention protocols

(No timestamps in these protocols)

Let transaction T try to lock an item that is already locked by T'

- no-waiting rule:
  - T aborts and restarts

Waiting-based deadlock prevention protocols

(No timestamps in these protocols)

Let transaction T try to lock an item that is already locked by T'

- no-waiting rule:
  - T aborts and restarts
- cautious-waiting rule:
  - if T' is not waiting for anyone, then T waits (regularly checked)
  - otherwise T aborts and restarts

Any of these rules guarantees no deadlocks

(no cyclic waiting is possible)

Special care should be taken to avoid starvation

(Homework: What can be a solution?)

Waiting-based deadlock prevention protocols

(No timestamps in these protocols)

Let transaction T try to lock an item that is already locked by T'

- no-waiting rule:
  - T aborts and restarts
- cautious-waiting rule:
  - if T' is not waiting for anyone, then T waits (regularly checked)
  - otherwise T aborts and restarts

Any of these rules guarantees no deadlocks

(no cyclic waiting is possible)

Special care should be taken to avoid starvation

(Homework: What can be a solution?)

Easier, but even more unnecessary aborts

Deadlock detection

Deadlock happens when

- there is a cycle of transactions waiting for each other

We can detect a cycle and

- break the cycle by aborting one transaction (the victim)

Deadlock detection

Deadlock happens when

there is a cycle of transactions waiting for each other

We can detect a cycle and

break the cycle by aborting one transaction (the victim)

Can be done by maintaining the wait-for graph:

- nodes: active transactions
- (directed) edges: transactions waiting for transactions

A cycle in the wait-for graph means a deadlock

Deadlock detection

Deadlock happens when

there is a cycle of transactions waiting for each other

We can detect a cycle and

break the cycle by aborting one transaction (the victim)

Can be done by maintaining the wait-for graph:

- nodes: active transactions
- (directed) edges: transactions waiting for transactions

A cycle in the wait-for graph means a deadlock

May be resource-consuming to maintain

Victim selection should be done cautiously to avoid starvation

Timeouts

Timeout-based deadlock avoidance:

- fix a waiting timeout period in advance
- a transaction waits for at most the timeout period
  - after this abort and restart

Simple and easy to maintain

No guarantees (unknown overhead, possible starvation)

2. Concurrency via timestamps

Timestamp-based concurrency control: Idea

Reminder:

Transaction timestamp in a unique number $TS(T)$ assigned to each transaction T so that $TS(T') < TS(T)$ for each T' started before

We used timestamps for deadlock prevention in the lock-based concurrency control

Timestamp-based concurrency control: Idea

Reminder:
Transaction timestamp in a unique number $TS(T)$ assigned to each transaction T so that $TS(T') < TS(T)$ for each T' started before

We used timestamps for deadlock prevention in the lock-based concurrency control

We can use timestamps for concurrency control without locking:

- optimistically runs transactions until something goes wrong (i.e., a conflict is detected)
- rolls back one of the conflict transactions based on their timestamps (roll back is abort and restart with a new timestamp)

Timestamp-based concurrency control: Overview

Timestamp-based concurrency control protocols:

- all ensure serialisability

  the serial version has transactions arranged by their timestamps
- may not guarantee recoverability

  additional techniques needed to ensure this

  (e.g., form of deadlock-free locking)
- may not guarantee no starvation

  additional techniques needed to ensure this

We will consider several timestamp-based protocols

- basic: ensures serialisability, but not recoverability
- strict: ensures serialisability and strictness (so recoverability)
- basic/strict with Thomas write rule: less roll-backs

Item timestamp variables

Besides the timestamp of a transaction (see above),
the concurrency control subsystem (of each timestamp-based protocol) maintains two variables for each item X:

- read timestamp $ReadTS(X)$:
  the timestamp of the youngest transaction that has read X
- write timestamp $WriteTS(X)$:
  the timestamp of the youngest transaction that has written X

Maintained in appropriate timestamp tables
(similar to the lock table in lock-based protocols)

(All initialised by 0)

Basic timestamp-ordering protocol: Idea

Main idea:

If there is a conflict that violates the order

(of the serial schedule imposed by the timestamps),

then roll back (i.e., abort and restart with new timestamp)

the transaction of the second operation of the conflict

Basic timestamp-ordering protocol: Idea

Main idea:

If there is a conflict that violates the order

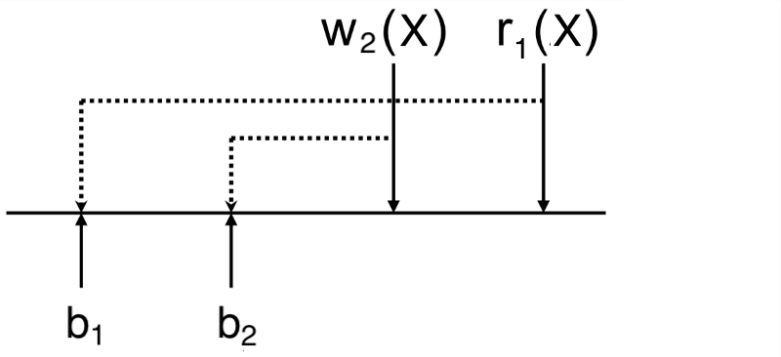
(of the serial schedule imposed by the timestamps),

then roll back (i.e., abort and restart with new timestamp)

the transaction of the second operation of the conflict

Read-write conflict 1:

a transaction may try to read data item X too late



Should abort and restart T_1

Basic timestamp-ordering protocol: Idea

Main idea:

If there is a conflict that violates the order

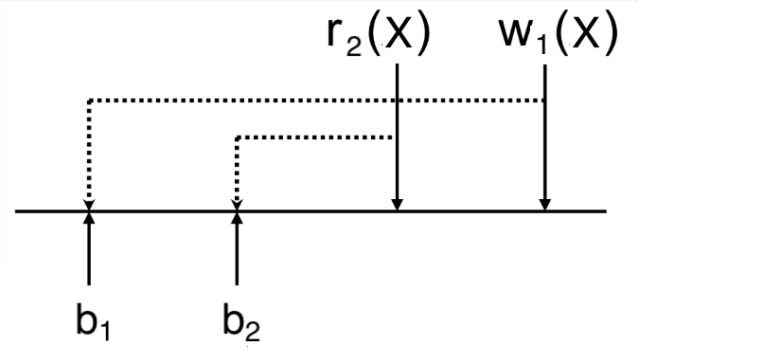
(of the serial schedule imposed by the timestamps),

then roll back (i.e., abort and restart with new timestamp)

the transaction of the second operation of the conflict

Read-write conflict 2:

a transaction may try to write data item X too late



Should abort and restart T_1

Basic timestamp-ordering protocol: Idea

Main idea:

If there is a conflict that violates the order

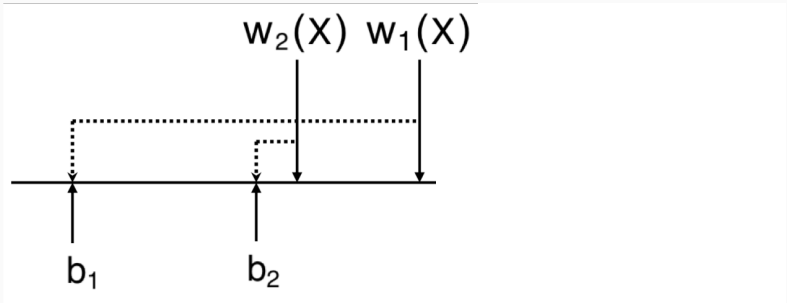
(of the serial schedule imposed by the timestamps),

then roll back (i.e., abort and restart with new timestamp)

the transaction of the second operation of the conflict

Write-write conflict:

a transaction may try to write data item X too late



Should abort and restart T_1 (or?...)

Basic timestamp-ordering protocol formally

Concurrency control subsystem should follow two rules:

Rule 1. When transaction T issues `read`(X):

- if $\mathit{WriteTS}(X) > \mathit{TS}(T)$
  (i.e., someone younger written X),
  then roll-back T (i.e., abort and restart T with a new timestamp)
- otherwise, execute the rest of `read`(X) and
  set $\mathit{ReadTS}(X)$ to $\max(\mathit{ReadTS}(X), \mathit{TS}(T))$

Rule 2. When transaction T issues `write`(X):

- if $\mathit{ReadTS}(X) > \mathit{TS}(T)$ or $\mathit{WriteTS}(X) > \mathit{TS}(T)$
  (i.e., someone younger accessed X),
  then roll-back T (i.e., abort and restart T with a new timestamp)
- otherwise, execute the rest of `write`(X) and
  set $\mathit{WriteTS}(X)$ to $\mathit{TS}(T)$

Basic timestamp-ordering protocol properties

Concurrency control subsystem should follow two rules:

Rule 1. When transaction T issues `read`(X):

- if $\mathit{WriteTS}(X) > \mathit{TS}(T)$ then roll-back T
- otherwise, execute `read`(X) and set $\mathit{ReadTS}(X)$ to $\max(\mathit{ReadTS}(X), \mathit{TS}(T))$

Rule 2. When transaction T issues `write`(X):

- if $\mathit{ReadTS}(X) > \mathit{TS}(T)$ or $\mathit{WriteTS}(X) > \mathit{TS}(T)$ then roll-back T
- otherwise, execute `write`(X) and set $\mathit{WriteTS}(X)$ to $\mathit{TS}(T)$

Ensures serialisable schedules, but they

- may be not recoverable,
- may have cascading roll-backs,
- may have starving transactions

Basic timestamp-ordering protocol properties

Concurrency control subsystem should follow two rules:

Rule 1. When transaction T issues `read`(X):

- if $\mathit{WriteTS}(X) > \mathit{TS}(T)$ then roll-back T
- otherwise, execute `read`(X) and set $\mathit{ReadTS}(X)$ to $\max(\mathit{ReadTS}(X), \mathit{TS}(T))$

Rule 2. When transaction T issues `write`(X):

- if $\mathit{ReadTS}(X) > \mathit{TS}(T)$ or $\mathit{WriteTS}(X) > \mathit{TS}(T)$ then roll-back T
- otherwise, execute `write`(X) and set $\mathit{WriteTS}(X)$ to $\mathit{TS}(T)$

Ensures serialisable schedules, but they

- may be not recoverable, (ensured by strict version)
- may have cascading roll-backs, (avoided by strict version)
- may have starving transactions (leave for you)

Reminder

To ensure durability in ACID, a schedule may be

- recoverable: T does not commit before each T' from which T read have committed
- cascadeless: T does not read from uncommitted transactions
- strict: T is cascadeless and does not overwrite any values written by uncommitted transactions

Strict timestamp-ordering protocol

Idea: In the rules the execution is postponed (using e.g., locking) until the pervious write is committed (so ensuring strict schedules)

Rule 1. When transaction T issues `read`(X):

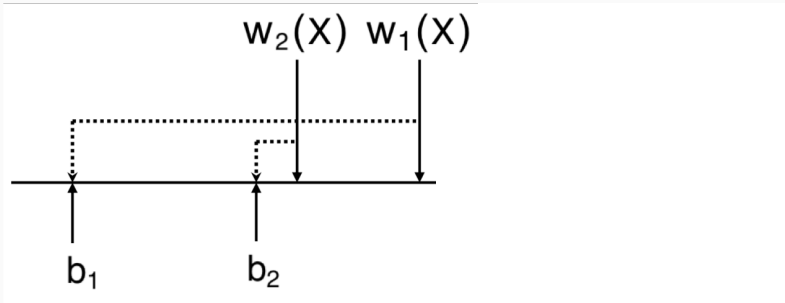
- if $\mathit{WriteTS}(X) > \mathit{TS}(T)$ then roll-back T
- otherwise
  - wait until the transaction T' such that $\mathit{WriteTS}(X) = \mathit{TS}(T')$ is committed (if there is such)
  - execute the rest of `read`(X) and set $\mathit{ReadTS}(X)$ to $\max(\mathit{ReadTS}(X), \mathit{TS}(T))$

Rule 2. When transaction T issues `write`(X):

- if $\mathit{ReadTS}(X) > \mathit{TS}(T)$ or $\mathit{WriteTS}(X) > \mathit{TS}(T)$ then roll-back T
- otherwise
  - wait until the transaction T' such that $\mathit{WriteTS}(X) = \mathit{TS}(T')$ is committed (if there is such)
  - execute the rest of `write`(X) and set $\mathit{WriteTS}(X)$ to $\mathit{TS}(T)$

Thomas writing rule

Improvement (less roll-backs) for both basic and strict protocols
in case of write-write conflict



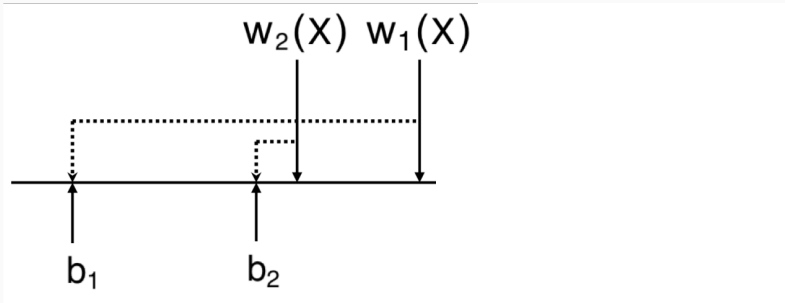
Rule 1. When transaction T issues `read`(X): ... (same)

Rule 2. When transaction T issues `write`(X):

- if $ReadTS(X) > TS(T)$ then roll-back T
- otherwise, if $WriteTS(X) > TS(T)$ ignore the operation and proceed
- otherwise ... (same, i.e., possibly wait, execute, update)

Thomas writing rule

Improvement (less roll-backs) for both basic and strict protocols
in case of write-write conflict



Rule 1. When transaction T issues `read`(X): ... (same)

Rule 2. When transaction T issues `write`(X):

- if $ReadTS(X) > TS(T)$ then roll-back T
- otherwise, if $WriteTS(X) > TS(T)$ ignore the operation and proceed
- otherwise ... (same, i.e., possibly wait, execute, update)

Same guarantees as the versions without the improvement

3. Multiversioning for concurrency control

Multiversioning concurrency control: Idea

Multiversioning idea:

several versions of the (value of the) same item
are kept by the system

- transactions are the same as usual
  (in particular, they request reads/writes for items, not versions)
- the versions of items are managed
  by the concurrency control subsystem

Multiversioning concurrency control: Idea

Multiversioning idea:

several versions of the (value of the) same item
are kept by the system

- transactions are the same as usual
  (in particular, they request reads/writes for items, not versions)
- the versions of items are managed
  by the concurrency control subsystem

Pros: increase concurrency (we will see how)

Cons: more storage is needed to maintain multiple versions

Multiversioning for concurrency control: Overview

Multiversioning: several versions of the same item are kept

Technical difficulty (for us):

Intuitively, still want to ensure semantic equivalence to a serial schedule,
but our notion of a schedule does not cover multiple versions of an item,
and all derived notions (conflicts, serialisability, etc.) are not formally applicable

Multiversioning for concurrency control: Overview

Multiversioning: several versions of the same item are kept

Technical difficulty (for us):
Intuitively, still want to ensure semantic equivalence to a serial schedule,
but our notion of a schedule does not cover multiple versions of an item,
and all derived notions (conflicts, serialisability, etc.) are not formally applicable

All these notions can be generalised appropriately (but clumsy)
So we assume such generalisation silently
Aiming for semantic serialisability of the multiversion schedule

We will (briefly) consider several multiversion protocols, based on

- timestamps
- two-phase locking
- validation
- snapshot isolation

Timestamps and timestamp variables for multiversioning

Concurrency control subsystem maintains:

- a timestamp $TS(T)$ of each transaction T (as before)
- several versions $X_0, X_1, \ldots$ of each data item X

  each version is written (created) only once
- two variables for each version X_i of each item:
  - read timestamp $ReadTS(X_i)$:

    the timestamp of the youngest transaction that has read X_i
  - write timestamp $WriteTS(X_i)$:

    the timestamp of the (only) transaction that has written X_i

Maintained in appropriate tables

(similar to the lock table in lock-based protocols)

(Each X_0 is initialised by the original value with 0 timestamps, at the end the last version is considered final)

Multiversion timestamp-ordering protocol

Main idea:

- A transaction reads the version written by the youngest transaction among those that are older than the reading transaction (always allowed)
- A transaction is allowed to write (create a new version) only if no one yet read a wrong value

Formally, concurrency control subsystem should follow two rules

Rule 1. When transaction T issues `read`(X):

- find the version X_i with the highest $WriteTS(X_i)$ less or equal than $TS(T)$ (if there are several versions with the same timestamp, take the one with the largest i)
- return this version to T and
  set $ReadTS(X_i)$ to $\max(ReadTS(X_i), TS(T))$

Multiversion timestamp-ordering protocol

Main idea:

- A transaction reads the version written by the youngest transaction among those that are older than the reading transaction (always allowed)
- A transaction is allowed to write (create a new version) only if no one yet read a wrong value

Formally, concurrency control subsystem should follow two rules

Rule 2. When transaction T issues `write`(X):

- find version X_i with highest $WriteTS(X_i)$ less or equal than $TS(T)$
- if $ReadTS(X_i) > TS(T)$ (i.e., someone younger accessed X), then roll-back T (i.e., abort and restart T with a new timestamp)
- otherwise, create a new version X_j and set both $ReadTS(X_j)$ and $WriteTS(X_j)$ to $TS(T)$

Multiversion timestamp-ordering protocol: Properties

Main idea:

- A transaction reads the youngest version among those that are created by older transactions (always allowed)
- A transaction is allowed to write (create a new version) only if no one yet read a wrong value

Ensures serialisable schedules (with the timestamp-ordered serial schedule), but they may

- be not recoverable,
- have cascading roll-backs,
- have starving transactions

Multiversion timestamp-ordering protocol: Properties

Main idea:

- A transaction reads the youngest version among those that are created by older transactions (always allowed)
- A transaction is allowed to write (create a new version) only if no one yet read a wrong value

Ensures serialisable schedules (with the timestamp-ordered serial schedule), but they may

- be not recoverable, (ensured by strict version)
- have cascading roll-backs, (avoided by strict version)
- have starving transactions

The strict (hence recoverable and cascadeless) version may be designed in the same way as for usual timestamp-based protocol

Multiversion two-phase locking protocol: Idea (only)

Maintains two versions of each data item:

- committed:
  - written by a committed transaction (or original)
  - all transactions (except the currently writing) read this
- local:
  - written by a currently writing transaction (only one is allowed)
  - visible for reading only for the writing transaction
  - rewrites (under certify lock) the committed version when the writing transaction commits

Three types of locks (read, write, certify) with compatibility table

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Validation-based protocol: Idea

Validation-based protocol:

- an optimistic serialisation strategy based on time stamping

  ('optimistic' is 'evaluate transaction and check serialisability after')

- each transaction has three phases:
  - read phase (includes all calculation)
  - validation phase (few checks, can roll back)
  - write phase

  (equivalent to keeping multiple local versions of items)

- the concurrency subsystem stores information

  only about transactions (not about items)

- aims for the serialisable schedule equivalent to the serial one

  according to the order of (completion of) validation

- (may be not recoverable and starving,

  need separate techniques to avoid them)

Snapshot isolation: Idea

Snapshot Isolation concurrency control protocol

- used in several practical DBMSs (PostgreSQL, Oracle, etc.)
- easy to ensure for the manager
- each transaction evaluates on the data version committed by the beginning of the transaction (i.e., multiversion)
- write-sets of interleaving transactions are disjoint
- does not guarantee serialisability

Snapshot isolation: Idea

Snapshot Isolation concurrency control protocol

- used in several practical DBMSs (PostgreSQL, Oracle, etc.)
- easy to ensure for the manager
- each transaction evaluates on the data version committed by the beginning of the transaction (i.e., multiversion)
- write-sets of interleaving transactions are disjoint
- does not guarantee serialisability

Example: $\mathtt{r}_1(X), \mathtt{r}_1(Y), \mathtt{r}_2(X), \mathtt{r}_2(Y), \mathtt{w}_1(Y), \mathtt{w}_2(X), \mathtt{c}_1, \mathtt{c}_2$

On the one hand, not (conflict-)serialisable

On the other hand, snapshot-isolated in essence:

- both transactions read only initial data
- write-sets are disjoint

4. Concurrency with multiple granularity

Data item granularity

Granularity: the size of data items

- field value of a record (fine)
- record
- disk block
- table file
- whole database (coarse)

Data item granularity

Granularity: the size of data items

- field value of a record (fine)
- record
- disk block
- table file
- whole database (coarse)

The tradeoff:

- more coarse allows for lower degree of concurrency
- more fine leads to more overhead (lock management, etc.)

Best item size depends on transaction type:

- if a transaction accesses few records, a record as item is good
- if a transaction accesses many records, a block as item is good

Data item granularity

Granularity: the size of data items

- field value of a record (fine)
- record
- disk block
- table file
- whole database (coarse)

The tradeoff:

- more coarse allows for lower degree of concurrency
- more fine leads to more overhead (lock management, etc.)

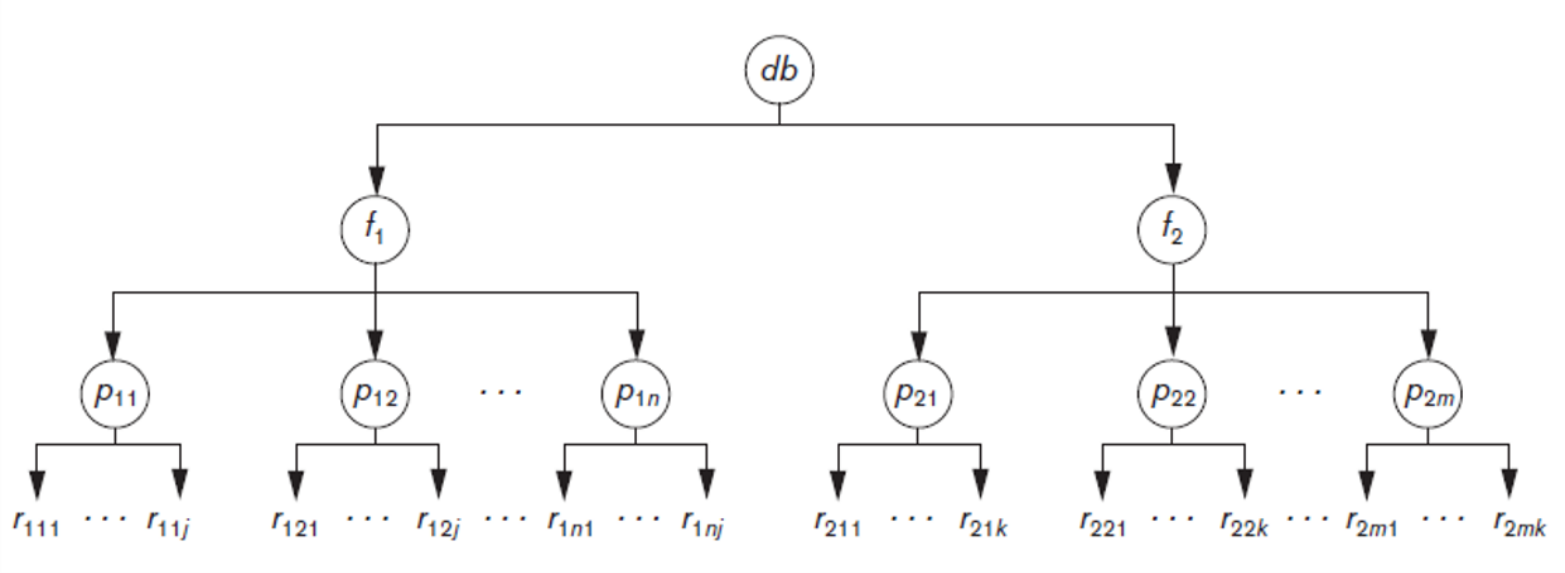
Best item size depends on transaction type:

- if a transaction accesses few records, a record as item is good
- if a transaction accesses many records, a block as item is good

It may be nice to have different granularity for different transactions

Multiple granularity locking: Example

Example granularity hierarchy:

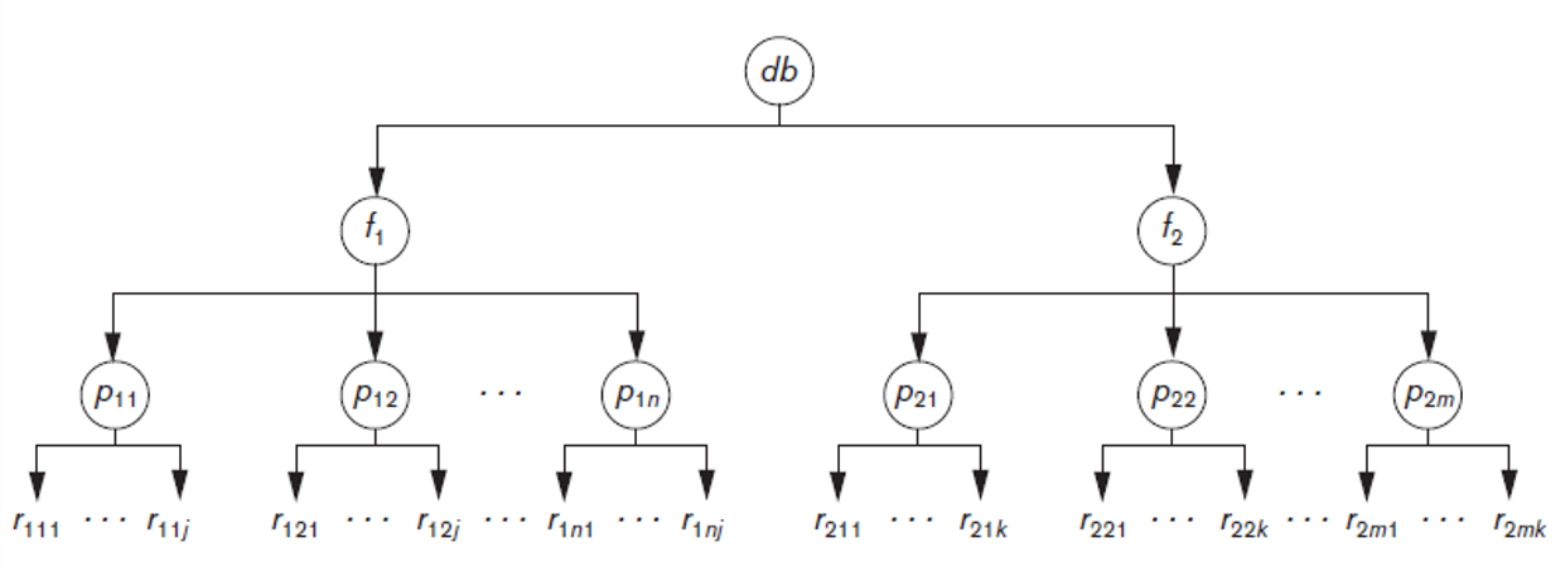


Let T_1 write-lock f_1 and then T_2 tries to read-lock r_{121}:

easy to detect and delay T_2 by traversing from leaf to root

Multiple granularity locking: Example

Example granularity hierarchy:



Let T_2 read-lock r_{121} and then T_1 tries to write-lock f_1:
detecting requires to search through the whole sub-tree,
which may be not efficient

Intention locks

Intention locks solve the problem:

besides usual (exclusive or shared) locks,
the locking transaction labels the path
from the root to the needed node with intension locks

- Intention-shared lock (IS):
  a shared lock is to be requested on a descendant node
  (one or several)
- Intention-exclusive lock (IX):
  an exclusive lock is to be requested on a descendant node
- Shared-intension-exclusive lock (SIX):
  current node is shared-locked and
  an exclusive lock is to be requested on a descendant node

Multiple-granularity two-phase locking

Multiple-granularity two-phase locking protocol rules (informally):

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

- each locking starts in the root and ends in the locked node
- the path to the node is locked with intentional versions and the node itself with the usual lock
- the lock-compatibility table (above) must be always adhered (wait if 'no')
- unlocking is symmetric, upgrading is possible

5. Other issues and conclusion

Other issues

There are many more issues with concurrency control:

- indexes, in particular $B(+)$-trees
  - using multiple-granularity two-phase locking is possible
  - not very efficient: insertion write-locks all the branch
  - dedicated approaches are used
    - (using the fact that all insertions start from the root)
- insertion and deletion may require special care
  - (write locks, etc.)
- phantom tuples may require special treatment
  - (when a new tuple is inserted in the middle of an affected selection by another transaction)
  - our protocols do not cover them
- interactive transactions
  - (something is shown to a user, but then should be rolled back)

What have we learned?

Concurrency control protocols

that ensure isolation via (conflict-)serialisability:

- locking-based

  (two-phase with variations,
  based on binary and shared/exclusive locks
  with several deadlock-avoidance methods)
- timestemp-based

  (basic, strict, Thomas write rule)
- multiversion-based

  (timestamp, two-phase, validation, snapshot)
- multiple-granularity-based

What have we learned?

Concurrency control protocols

that ensure isolation via (conflict-)serialisability:

- locking-based
  (two-phase with variations,
  based on binary and shared/exclusive locks
  with several deadlock-avoidance methods)
- timestemp-based
  (basic, strict, Thomas write rule)
- multiversion-based
  (timestamp, two-phase, validation, snapshot)
- multiple-granularity-based

In the rest of Part 2 (next week):

- techniques for recovery