Concurrency Control Algorithms

Concurrency Control Techniques

There are three basic concurrency control techniques that are used to ensure isolation of transactions:

1. ***Locking algorithms*** – to ensure serializability this approach employs the technique of locking data items to prevent transactions from accessing the data items concurrently
   - two-phase locking algorithm (2PL)

2. ***Timestamp ordering algorithms*** – to ensure serializability this approach employs timestamps; the serializability order corresponds to the order of transaction timestamps.

3. ***Optimistic algorithms*** – to ensure serializability this approach apply the concept of validation of a transaction after it executes its operations.

Locking techniques for concurrency control

A ***lock*** is a variable associated with a data item in the database and describe the status of that data item with respect to possible operations that can be applied to the item.

Generally, there is one lock for each data item.

A data item can have three states:
- data item is ***non-locked 0***
- data item is ***read-locked R*** (share-locked ***S***)
- data item is ***write-locked W*** (exclusive-locked ***X***)

Three additional operations must be included in the transaction when locking is used:

- Read-lock data item (LR(x))
- Write-lock data item (LW(x))
- Unlock data item (UNL(x))

The above operations must be implemented as indivisible units.

Compatibility of locks

Two lock requests are ***compatible*** if they allow concurrent access to the same data item.

lock requested \ current lock	***R***	***W***
R	✓	–
W	–	–

Convertibility of locks

lock requested \ current lock	***R***	***W***
R	✓	–
W	✓	✓

- transaction upgrade the lock
- transaction downgrade the lock

Locking implementation

Data structure:

data	***tid***	***lock***
x_1	T_1	**W**
x_2	T_1	**R**
x_2	T_2	**R**
x_3	T_2	**W**

data	***tid***	***lock***	***queue***
x1	T_2	**R**	1
x1	T_3	**W**	2
x2	T_4	**W**	1

Operations: ***LOCK***, ***R_lock***, ***W_lock***, ***Unlock***

LOCK*(X, tid)* $\rightarrow$ *{**0**, **R**, **W**}*

```
R_lock(X, tid) begin
  B: if (LOCK(X, tid) = 0 or LOCK(X, tid) = R)
     then LOCK(X, tid) ← R;
     else begin
        < insert into queue(X) and wait until lock manager
                                   wakes up the transaction>;
        go to B;
     end;
end R_lock;
```

```
W_lock(X, tid) begin
  B: if LOCK(X, tid) = 0
     then LOCK(X, tid) ← W;
     else begin
        < insert into queue(X) and wait until lock manager
                                   wakes up the transaction>;
        go to B;
     end;
end W_lock;
```



```
Unlock(X, tid) begin
  if LOCK(X, tid) = W
    then begin
      LOCK(X, tid) ← 0;
      < wake up one of the waiting transactions, if any >;
    end;
    else if LOCK(X, tid) = R
      then begin
        LOCK(X, tid) ← 0;
        if (number_of_read_locks_on_X = 0) then
          begin
            < wake up one of the waiting transactions, if
              any >;
          end;
      end;
end Unlock;
```

Locking algorithms

T_1	T_2
R_lock(T_1, Y)	*R_lock(T_2, X)*
read(Y)	*read(X)*
unlock(Y)	*unlock(X)*
W_lock(T_1, X)	*W_lock(T_2, Y)*
read(X)	*read(Y)*
X := X + Y;	*Y := Y + X;*
write(X);	*write(Y);*
unlock(X)	*unlock(Y)*

Initial values:
X = 20, Y = 30

Results of serial schedules of transactions T_1 and T_2 :
($T_1 \rightarrow T_2$) : X = 50, Y = 80; ($T_2 \rightarrow T_1$) : X = 70, Y = 50;

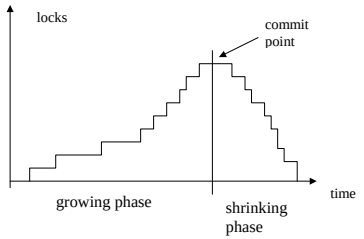
Using locks in transactions does not guarantee serializability of schedules in which the transaction participate.

Concurrent schedule:

T_1	T_2
R_lock(T_1, Y)	
read(Y)	
unlock(Y)	
	R_lock(T_2, X)
	read(X)
	unlock(X)
	W_lock(T_2, Y)
	read(Y)
	Y := Y + X;
	write(Y);
	unlock(Y)
W_lock(T_1, X)	
read(X)	
X := X + Y;	
write(X);	
unlock(X)	

Result of the schedule:
X = 50, Y = 50
(nonserializable schedule)

Two phase locking algorithm



Basic algorithm:

1. A transaction T must issue the operation *R_lock(X, T)* or *W_lock(X, T)* before any *read(X)* operation is performed in T
2. A transaction T must issue the operation *W_lock(X, T)* before any *write(X)* operation is performed in T
3. A transaction T must issue the operation *unlock(X, T)* after all *read(X)* and *write(X)* operations are completed in T

Static algorithm: (1., 2., 3.)

4. A transaction has to lock all data items it accesses before the transaction begins execution by predeclaring ikts read and write set.

Example:

T_1	T_2
R_lock(T_1, Y)	
read(Y)	
W_lock(T_1, X)	
	R_lock(T_2, X)
read(X)	*(wait)*
X := X + Y;	*(wait)*
write(X);	*(wait)*
unlock(Y)	*(wait)*
unlock(X)	*(wait)*
	read(X)
	W_lock(T_2, Y)
	read(Y)
	Y := Y + X;
	write(Y);
	unlock(X)
	unlock(Y)

Deadlock

T_1	T_2
R_lock(T_1, Y)	
read(Y)	
	R_lock(T_2, X)
	read(X)
	W_lock(T_2, Y)
W_lock(T_1, X)	*(wait)*
(wait)	*(wait)*
(wait)	*(wait)*
dead	***lock***

Possible approaches:

- deadlock prevention
- deadlock detection

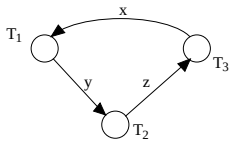
Deadlock prevention

Deadlock prevention protocols use the concept of transaction timestamp - *TS(T)*, which is unique identifier assigned to each transaction at its initiation. Suppose that transaction T_i tries to lock an item X, but is not able to because X is already locked in incompatible mode by some other transaction T_j. The rules followed by these protocols are as follows;

- ***wait-die***:
  if $TS(T_i) < TS(T_j)$ (T_i is older than T_j)
  then T_i is allowed to wait;
  otherwise abort T_i and restart it later with the same timestamp
- ***wound-wait***:
  if $TS(T_i) < TS(T_j)$ (T_i is older than T_j)
  then abort T_j (T_i wounds T_j) and restart it later with the same timestamp
  otherwise T_i is allowed to wait;

Deadlock detection

Waits-For Graph



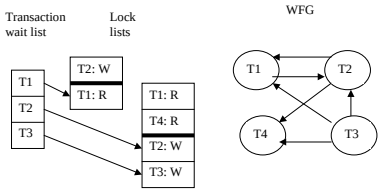
Deadlocks tend to be rare and typically involve very few transactions. So, it may be better to detect and solve deadlocks aa they arise.

- The WFG is periodically checked for cycles. A deadlock is resolved by aborting a transaction that is on a cycle and releasign its locks
- The ***timeout mechanism***: if a transaction has been waiting too long for a lock (miss the timeout), we can assume that it is deadlock cycle and resolve it.

Deadlock Detection Procedure

The lock data structures can be used to build the WFG. Each lock L has two lists, the granted list and the waiting list. Both lists have the form ((Ti, mi),...), where each Ti is a transaction and each mi is a lock mode. The edge Ti → Tj should be added to the WFG if:

- transaction Tj is in granted list and transaction Ti is in waiting list
- transaction Tj is ahead of Ti in waiting list, and
- modes mi and mj are incompatible.



No waiting approach to deadlock

Another group of protocols that prevent deadlock do not require timestamps:

- ***no waiting (NW)***: if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not.
- ***cautious waiting (CW)***: A transaction tries to lock a data item but is unable to obtain a lock. If T_j is not blocked (not waiting for some other locked data items) then T_i is blocked and allowed to wait, otherwise abort T_i.

No waiting approach may lead to the following phenomena:

- Dynamic deadlock (livelock)
- Infinite restarting

Additional Issues

The Convoy Phenomenon:

A transaction T holds a heavily used lock. An operating system with preemptive scheduling startegy will suspend T. Until T is resumed, every other transaction that needs this lock is queued. Such queue is called convoy.
Convoy can become very long and once formed tends to be stable. Convoys are one of the drawbacks of building a DBMS on top of a general-purpose operating system with preemptive scheduling.
(see Log manager)

Latches

Each DBMS support short-duration locks called latches. Setting a latch before reading or writing a page ensures that the physical read or write operation is atomic; otherwise two read/write operations may conflict if the object being locked do not correspond to disk pages. Latches are unset immediately after the physical read or write operation is completed.

Phantom Problem

We have considered a database as a set of a fixed and independent data objects.

```
select *
from emp
where eyes="blue" and hair="red";

delete from emp
where eyes="blue" and hair="red";
```

Suppose an individual record-locking scheme is used.

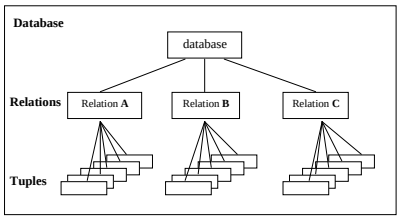
- How to prevent someone else from inserting a new, blue-eyed, red-haired emp?
- It is easy to demonstrate that concurrent execution of two transactions is not serializable.

Such new or deleted records are called ***phantoms*** – records that either appear or disappear from relations.

There is no pure record-locking solution for phantoms

Multiple-Granularity Locking

- A lock hierarchy



Phantoms raise the issue of the ***lock unit***, the data aggregates that are locked to insure isolation (databases, relations, tuples, fields, etc.):

Lock Unit

The choice of lock granule presents a trade-off between concurrency and overhead:

- Concurrency control is maximized by a fine locking granule (record-level locking)
- A fine locking is costly for a complex transactions accessing a large number of granules – the large transaction would have to acquire and maintain a lock on each granule
- A corse locking granule (e.g.) relation is convenient for complex transactions, but it would discriminate the small transactions that want to access just a tiny part of the relation.

Conclusion: we need a locking protocol that satisfy all these needs; it should let batch transactions set a single lock that covers an entire relation, while letting small interactive transactions lock finer granules.

The idea is to use a fixed set of predicate locks combined with the hierarchical lock protocol.

Intent lock modes

The idea of hierarchical lock protocol:

> Locking a node of the tree (a lock hierarchy) in some mode implicitly locks all the descendants of that node in the same locking mode.

- Locking "Relation A" in exclusive lock mode (write lock mode) implicitly locks in exclusive mode that relation as well as all tuples below the "Relation A" node in the lock tree.
- To lock a subtree rooted at a certain node without locking any of the other relations, a transaction must prevent a shared or exclusive lock from being set on the root node and on the "Relation A" node.

This is best done by inventing a new mode, called intent mode, which represents the intention to set locks at the finer granularity.

Intent lock modes

Intent lock modes:

- ***IR*** (IS) – a transaction intents to set shared (read) lock at finer granularity;
- ***IW*** (IX) - a transaction intents to set exclusive (write) or shared (read) lock at finer granularity;
- ***RIW*** (SIX) – a transaction intents to set a coarse granularity shared lock with intent to set finer granularity exclusive locks – essentially the union of R and IR.

Compatibility Matrix for Granular Locks

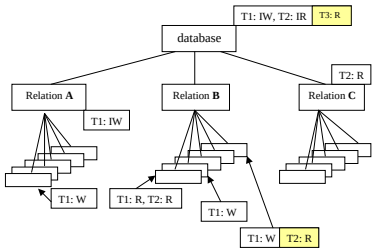
Requested lock \ Granted lock	IR	IW	R	RIW	U	W
IR	✓	✓	✓	✓	–	–
IW	✓	✓	–	–	–	–
R	✓	–	✓	–	–	–
RIW	✓	–	–	–	–	–
U	–	–	✓	–	–	–
W	–	–	–	–	–	–

Multiple Granularity Locking Protocol

1. Acquire locks from root to leaf
2. Release locks from leaf to root
3. To acquire an R mode or IR mode lock on a non-root node, one parent must be held in IR mode or higher (one of {IR, IW, RIW, W, U})
4. To acquire an W, U, RIW, or IW mode lock on a non-root node, all parents must be held in IW mode or higher (one of {IW, RIW, W, U})

Example:

Three transactions operating on a lock tree using different lock modes.



Update Mode Locks

Update mode locks are introduced to avoid the most common form of deadlock.

```
update emp
set salary=salary*1.1
where name="Morzy";
```

- A shared lock on a record is acquired to cover the read, and then an exclusive lock is requested to cover subsequent write

- The problem of ***hotspots*** – frequently updated records
- The study of System R showed that virtually all deadlocks in that system were of this form
- Acquiring a lock in ***Update lock mode*** avoids this read-write form of deadlock.
- Update lock mode is asymmetric

Multiple Granularity Locking Summary

Six lock modes are defined:

- **Intention read (IR)** – gives the grantee authority to explicitly set IR and R mode locks at a finer granularity, and prevents others from holding write (W, U) on this node
- **Intention write (IW)** – gives the grantee authority to explicitly set IR, IW, R, RIW, U, and W mode locks at a finer granularity, and prevents others from holding coarse granularity (R, RIW, W, U) locks on this node
- **Read and intention write (RIW)** – gives the grantee read authority to the node and its descendants (the equivalent of read authority), and prevents others from holding coarse or update locks (W, U, IW, RIW, R) on this node or its descendants. In addition, it gives the grantee authority to explicitly set IW, U, and W mode locks at a finer granularity.
- **Read (R)** – gives the grantee read authority to the node and its descendants, and prevents others from holding update mode locks (IW, W, RIW) on this node or its descendants
- **Update (U)** – gives the grantee read authority to the node and its descendants, and prevents others from holding nonshared locks (IW, W, RIW, U, IR) on this node or its descendants. This mode represents an intention to update the node in the future. To prevent a

common form of deadlock, it is not compatible with itself.
- **Write (W)** – gives the grantee write authority to the node, and prevents others from holding locks (W, U, R, RIW, IR, IW) on this node or its descendants. In addition, it gives the grantee authority to explicitly set any mode of lock at a finer granularity.

Timestamp Ordering Concurrency Control

In lock based concurrency control, conflicting operations of different transactions are ordered by the order in which locks are obtained. The lock protocol extends this ordering on operations to transactions thereby ensuring serializability.

In timestamp ordering concurrency control, a timestamp ordering is imposed on transactions and concurrency control algorithm checks that all conflicting operations occurred in the same order.

Each transaction T is assigned ***a timestamp*** TS(T) at the startup. The algorithm ensures, at execution time, that if action ai of transaction Ti conflicts with action aj of transaction Tj, ai occurs before aj if TS(Ti) < TS(Tj). If an action violates this ordering,the transaction is aborted and restarted with a new timestamp.

Timestamp Ordering Concurrency Control (cont)

Moreover, each data item x is given two timestamps:

- ***Read_TS(x)*** – maximum timestamp of a transaction that read the data item x.
- ***Write_TS(x)*** – timestamp of a transaction that write the data item x.

Implementation of the timestamp ordering algorithm:

Read procedure:

Read(T_i, x) *begin*
  *if (**TS**(T_i) < **Write_TS**(x)) then*
    < abort T_i and restart it with a new timestamp >;
  else begin
    < read x >;
    ***Read_TS**(x) ← max (**Read_TS**(x), **TS**(T_i));*
  end;
*end **Read**;*

Write procedure:

Write(T_i, x) *begin*
  *if (**TS**(T_i) < **Read_TS**(x) or **TS**(T_i) < **Write_TS**(x)) then*
    < abort T_i and restart it with a new timestamp >;
  else begin
    < write x >;
    ***Write_TS**(x) ← **TS**(T_i);*
  end;
*end **Write**;*

- The algorithm is deadlock-free

S: T1: r(x) T2: r(y) T1: w(y) T2: w(x) T1: c T2: c

- The algorithm does not provide recoverability of schedules!!!

S: T1: w(x) T2: r(x) T2: w(x) T2: c T1: abort

Timestamp Ordering - buffering of operations

To ensure the recoverability of schedules produced by T/O algorithm it is necessary to modify the protocol by buffering all write operations until the transaction commits.

- When a transaction T1 wants to write x, Write_TS(x) is updated to reflect this action, but the change to x is not carried out immediately; instead it is recorded in a private workspace (buffer)
- When T2 wants to read x subsequently, its timestamp TS(T2) is compared with Write_TS(x), and the read is seen to be permissible – however, T2 is buffered until T1 is completed (commited).

The buffering is similar to the effect of T1 obtaining an exclusive lock on x!!!

Optimistic Concurrency Control

- Locking concurrency control take a pessimistic approach to conflicts between transactions – so they use either transaction abort or blocking to resolve conflicts.
- In systems with relatively light contention for data items, the overhead of obtaining locks and following a locking algorithm is pretty high.
- In optimistic concurrency control, the basic assumption is that most transactions will not conflict with other transactions, and the idea is to be as permissive as possible in allowing transactions to execute.

Optimistic Concurrency Control Phases

Transactions proceed in three phases:

- ***Read phase***: The transaction executes, reading values from the database and writing new values into a private workspace.
- ***Validation phase***: If the transaction decides to commit, the optimistic algorithm checks whether the transaction could possibly have conflicted with any other concurrently executing transaction. If there is a possible conflict, thansaction is aborted; its private workspace is cleared and it is restarted.
- ***Write phase***: If validation determines that there are no possible conflicts, the changes to data items made by the transaction in its private workspace are copied into the database.

Validation Procedure

Each transaction Ti is assigned a timestamp TS(Ti) at the beginning of its validation phase, and the validation procedure checks whether the timestamp ordering of transactions is an equivalent serial order.

Moreover, for each transaction Ti is kept an interval of timestamps (TS_{start}, TS_{finish}) that defines transactions entering the validation phase while the transaction Ti is in its read phase. The set of transactions with timestamps belonging to the above interval is denoted as $Tr\_set_{active}(Ti)(TS_{start}+1, TS_{finish})$. Moreover, for each transaction Ti two sets of data items are kept: fr(ti) and fw(Ti), which denote respectively a set of data items read (written) by Ti.

For every pair of transactions Ti and Tj such that TS(Ti) < TS(Tj), one of the following conditions must hold:

1. Ti completes (all three phases) before Tj begins; or
2. Ti completes before Tj starts its Write Phase, and Ti does not write any data item that is read by Tj; or

3. Ti completes its Read Phase before Tj completes its Read Phase, and Ti does not write any data item that is ejther read or written by Tj.

To validate Tj we must check to see that one of these conditions holds with respect to each committed transaction Ti such that TS(Ti) < TS(Tj).

Validation Phase

Procedure Validate(Ti)
test = false
for each $Tj \in Tr\_set_{active}(Ti)(TS_{start}+1, TS_{finish})$
  if $(fr(Ti) \cup fw(Ti)) \cap fw(Tj) \neq \varnothing$
  then test = true
if test **then**
<write phase>
  write fw(Ti)
  increase **Global Counter** (GC) by 1
</end write phase>
else abort Ti

Degrees of Isolation

Every transaction has three characteristics: **diagnostics_size**, **access_mode**, and **isolation_level**.

Diagnostics_size:

The diagnostics_size determines the number of error condition that can be recorded for the transaction.

Access_mode:

There are two access_modes: READ ONLY and READ WRITE.

If the access_mode is READ ONLY, the transaction is not allowed to modify the database. Thus INSERT, DELETE, UPDATE and CREATE statements cannot be executed. For transactions with READ ONLY access_mode, only shared **locks need to be** obtained, thereby increasing concurrency.

If we have to execute one of commands INSERT, DELETE, UPDATE or CREATE, the access_mode should be set to READ WRITE.

Isolation_levels

Most systems do not provide automatically serializability!!

- Implementors did not understand the issues
- Implementors make a compromise between correctness and performance and provide options called ***levels of isolation*** (or degrees of isolation)

The isolation_level controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation_level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transaction's uncommitted changes.

In SQL-92 the isolation levels are:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

Isolation_levels

- SERIALIZABLE – this isolation level ensures that T reads only the changes made by committed transactions, that no value read or writtten by T is changed by any other transaction until t is complete.

  In terms of lock-based implementation, a SERIALIZABLE means that the lock algorithm is two-phase and well formed.

- REPEATABLE READ - this isolation level ensures that T reads only the changes made by committed transactions, that no value read or writttten by T is changed by any other transaction until t is complete. However, T could experience the phantom phenomenon.

  In terms of lock-based implementation, a REPEATABLE READ means that the lock algorithm is two-phase and well formed.
  A REPEATABLE READ uses the same locking protocol as a SERIALIZABLE transaction except that it does not do index locking – it locks only individual objects - not sets of objects.

Isolation levels

- READ COMMITTED (*cursor stability*) - this isolation level ensures that T reads only the changes made by committed transactions, that no value writtten by T is changed by any other transaction until t is complete. However, a value read by T may well be modified by another transaction while T is in progress, and T is exposed to the phantom phenomenon.

  In terms of lock-based implementation, a READ COMMITTED means that the lock algorithm is two-phase with respect to write locks and well formed with respect to reads. In other words, all shared locks obtained by T are released immediately.

- READ UNCOMMITTED (browse)- T can read changes made to an object by an ongoing transaction. Moreover, the object can be changed further while T is in progress, and T is exposed to the phantom phenomenon.

  In terms of lock-based implementation, a READ UNCOMMITTED means a transaction T obtains write locks before writing data items, and holds these locks until the end, but does not obtain shared locks before reading data items.

  READ UNCOMMITTED is allowed only for read-only transactions – a transaction is required to have an access mode of READ ONLY.

Isolation levels

Isolation Level	**Dirty Read**	**Unrepeatable Read**	**Phantom**
READ UNCOMMITTED	maybe	maybe	maybe
READ COMMITTED	no	maybe	maybe
REPEATABLE READ	no	no	maybe
SERIALIZABLE	no	no	no

Why READ COMMITTED is called sometimes *Cursor Stability*?

```
exec sql select balance
        into :balance
        from account
        where account_id=:id;

balance=balance+10;

exec sql update account
        set balance=:balance
        where account_id=:id;
```



```
exec sql declare cursor c for
        select balance
        from account
        where account_id=:id;

exec sql open cursor c
exec sql fetch c into :balance
balance=balance+10;

exec sql update account
        set balance=:balance
        where current of cusor c;
exec sql close c;
```

- Most SQL-systems keep a shared lock on the record currently addressed by a cursor.

The isolation and access-mode can be set using the SET TRANSACTION command. The following command declares the current to be SERIALIZABLE and READ ONLY:

```
SET TRANSACTION ISOLATION LEVEL
SERIALIZABLE READ ONLY
```