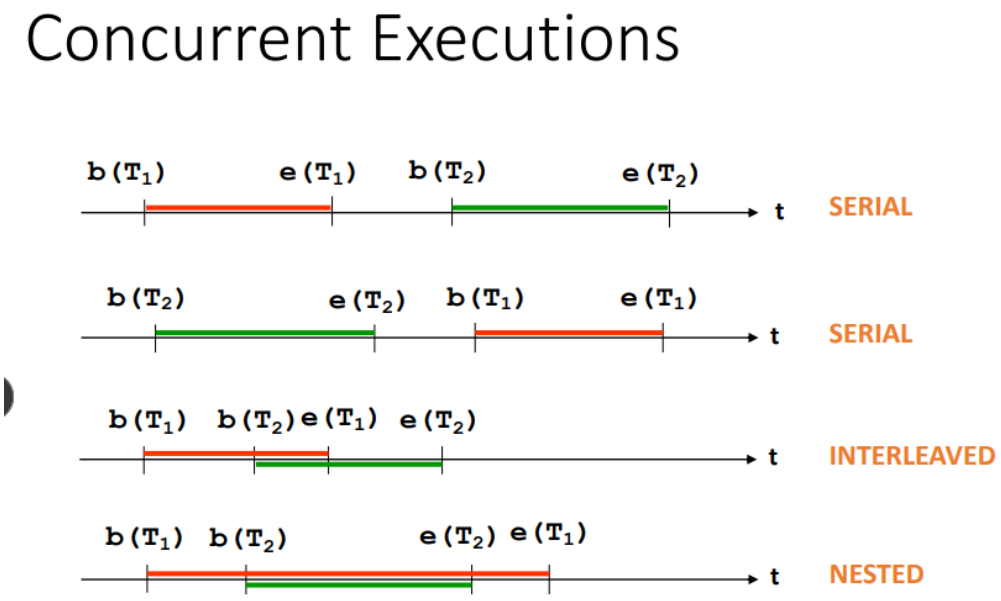
02.Concurrency Control

The Concurrency Control System manages the simultaneous execution of transactions,avoids the insurgence of anomalies while ensuring performance.
One of the advantages of concurrency is to exploit the parallelism to maximise transactions per second(TPS).
The problems of concurrency is that the transactions have to be executed in parallel as they were done in sequence.
Concurrency is fundamental as DB need to execute tens or hundreds of transactions per second and naturally they cannot be done serially, but the concurrent execution may cause anomalies $\rightarrow$ concurrency needs to be controlled



Execution with Lost Update

An update is applied from a state that ignores a preceding update, which is lost
D = 100
1 T_1: r(D $\rightarrow$ x)
2 T_1: x = x + 3
3 T_2: *r(D $\rightarrow$ y)*
4 T_2: *y = y + 6*
5 T_1: w(x $\rightarrow$ D) D = 103
6 T_2: *w(y $\rightarrow$ D) D = 106 !*
Note: this anomaly does not depend merely on T_2 overwriting the value produced by T_1

- w_1(D), w_2(D) is ok (serial)
- r_1(D), w_1(D), w_2(D) is ok too (serial)
- r_1(D), r_2(D), w_1(D), w_2(D) is not ok: inconsistent updates from the same

Dirty Read

An uncommitted value is used to update the data
D = 100
1 T_1: r(D $\rightarrow$ x)
2 T_1: x = x + 3
3 T_1: w(x $\rightarrow$ D) D = 103
4 T_2: r(D $\rightarrow$ y) this read is “dirty” (uncommitted value)
5 T_1: rollback
6 T_2: y = y + 6
7 T_2: w(y $\rightarrow$ D) D = 109 !

Non-Repeatable Read

Someone else updates a previously read value
D = 100
1 T_1: r(D $\rightarrow$ x)
2 T_2: r(D $\rightarrow$ y)
3 T_2: y = y + 6
4 T_2: w(y $\rightarrow$ D) D = 106
5 T_1: r(D $\rightarrow$ z) z <> x !

Phantom update

Someone else updates data that contributes to a previously valid constraint
Constraint: A+B+C=100, A=50, B=30, C=20
T_1: r(A $\rightarrow$ x), r(B $\rightarrow$ y)……
T_2: r(B $\rightarrow$ s), r(C $\rightarrow$ t)
T_2: s = s + 10, t = t – 10
T_2: w(s $\rightarrow$ B), w(t $\rightarrow$ C) (now B=40, C=10, A+B+C=100)
T_1: r(C $\rightarrow$ z) (but, for T_1, x+y+z = A+B+C = 90!)
So for T_1 it is as if “somebody else” had updated the value of the sum
But for T_2 the update is perfectly legal (does not change the value of the sum)

Phantom Insert

Someone else inserts data that contributes to a previously read datum
T_1: C=AVG(B: A=1)
T_2: Insert (A=1,B=2
T_1: C=AVG(B: A=1)
Note: this anomaly does not depend on data already present in the DB when T_1 executes, but on a “phantom” tuple that is inserted by “someone else” and satisfies the conditions of a previous query of T_1

Concurrency Theory vs System Implementation

Model: an abstraction of a system, object or process, which purposely disregards details to simplify the investigation of relevant properties
Concurrency Theory builds upon a model of transactions and concurrency control principles that helps understanding real systems
Real systems exploit implementation level mechanisms (locks, snapshots) which help achieve some of the desirable properties postulated by the theory:

- Do not look for a view or conflict serializability checker in your DBMS
- Look instead for lock tables, lock types, lock granting rules, snapshots, etc..
- Understand how the implementation mechanisms ensure properties modelled by the Concurrency Theory

Transactions, Operations, Schedules

Operation: a read or write of a specific datum by a specific transaction
Schedule: a sequence of operations performed by concurrent transactions **that respects the order of operations of each transaction**

Schedules

How many distinct schedules exist for two transactions?

$N = 6$

- With T_1 and T_2 from the previous slide:
- $r_1(x)$ $w_1(x)$ $r_2(z)$ $w_2(z)$ — *serial*
- $r_2(z)$ $w_2(z)$ $r_1(x)$ $w_1(x)$ — *serial*
- $r_1(x)$ $r_2(z)$ $w_1(x)$ $w_2(z)$ — *interleaved*
- $r_2(z)$ $r_1(x)$ $w_2(z)$ $w_1(x)$ — *interleaved*
- $r_1(x)$ $r_2(z)$ $w_2(z)$ $w_1(x)$ — *nested*
- $r_2(z)$ $r_1(x)$ $w_1(x)$ $w_2(z)$ — *nested*

We have N_S serial schedules for n transactions and N_D distinct transactions.

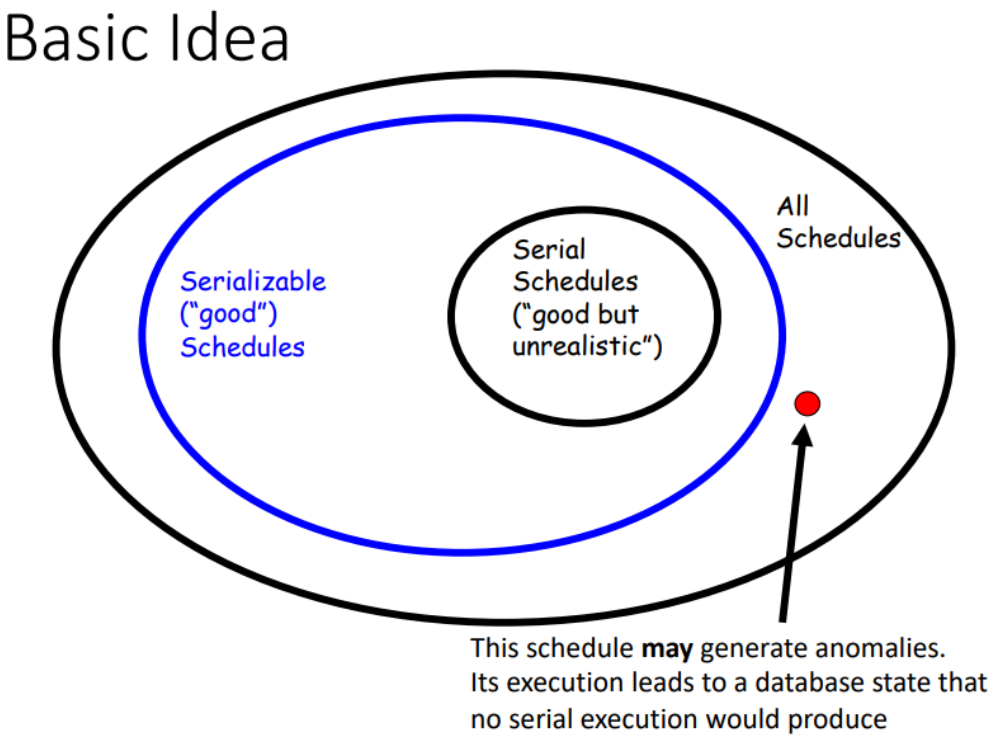
$$N_S = n!, N_D = \frac{(\sum_{i=1}^{n} k_i)!}{\prod_{i=1}^{n}(k_i!)}$$

Principles of Concurrency Control

Goal: to reject schedules that cause anomalies
Scheduler: a component that accepts or rejects the operations requested by the transactions
Serial schedule: a schedule in which the actions of each transaction occur in a contiguous sequence
Serializable schedule: a schedule that leaves the database in the same state as some serial schedule of the same transactions. Commonly accepted as a notion of schedule correctness.

Requires a notion of schedule equivalence to identify classes of schedules that ensure serializability
Assumption:

- We initially assume that transactions are observed “a- posteriori” and limited to those that have committed (commit-projection), and we decide whether the observed schedule is admissible
- In practice (and in contrast), schedulers must make decisions while transactions are running
  DB need to make decision when the commit is done.



View Serializability

Preliminary definitions:

- r_i(x) reads-from w_j(x) in a schedule S when w_j(x) precedes r_i(x) and there is no w_k(x) in S between r_i(x) and w_j(x)
- w_i(x) in a schedule S is a final write if it is the last write on x that occurs in S
  Given two schedule we want to decide if they are view-equivalent($S_i \approx_V S_j$):

1. If we have two transaction that works on the same resource they must have the same read-from relationship
2. They have the same operations
3. the same final writes
   A schedule is view-serializable if it is view-equivalent to a serial schedule of the same transactions. The class of view-serializable schedules is named VSR.
   Mnemonically: S is view-serializable if

4. every read operation sees the same values and
5. the final value of each object is written by the same transaction
   - as if the transactions were executed serially in some order
   The reads-from relationship r_i(x) reads-from w_j(x) in a schedule S assumes that r_i(x) reads the value written by w_j(x) independently of the time at which the commit of T_j occurs. In other words the value written by w_j(x) could be uncommitted when r_i(x) reads it but we are sure (by definition of commit projection) that it will be committed
   The check is linear. The complexitiy comes from the fact we have a factorial number of transaction we have to scan to determine if two transactions are view-serializable $\rightarrow$ it is a NP-Complete Problem. So we look for a stricter definition that is easier to check.

Conflict-serializability

A conflict is a situation where in a schedule I have two operations by two transaction in which at least one is a write on the same value of the other.Two operations o_i and o_j ($i \neq j$) are in conflict if
they address the same resource and at least one of them is a write

- read-write conflicts (r-w or w-r)
- write-write conflicts (w-w)
  Two schedules are conflict-equivalent ($S_i \approx_C S_j$) if :
- S_i and S_j contain the same operations and in all the conflicting pairs the transactions occur in the same order
- A schedule is conflict-serializable iff it is conflict-equivalent to a serial schedule of the same transactions
- The class of conflict-serializable schedules is named CSR
  CSR $\subset$ VSR: all conflict-serializable schedule are also view-serializable, but is not true the contrary.
  Conflict-equivalence implies view-equivalence therefore conflict-serializability implies view-serializability.

Testing Conflict-Serializability

It is done with a conflict graph: a simple directed graph with as many node as transactions and an ark for every conflict in the schedule(start from the first transaction and it arrive at the second transacction). This graph is CSR only if this graph is acyclic. It is done with $O(n^2)$ or $O(n^3)$ and for us it is acceptable as we are arriving from a NP-problem. The ark are referred to every conflict occurs on a resource even if the operations aren't consecutive.
We can prove that being in CSR it implies an acyclic conflict-graph and vice-versa.

Concurrency Control in Practice

The assumption of CSR and VSR is that they are working on what will be committed to the database, it is as we have everything beforehand that is unrealistic. Not feasible to do it at every

operation. We want to find a clever way to not waste so much time.
The interpretation of the schedule is a posteriori view. From now we will use an arrival sequence(sequence of request by transactions). From a notation point of view it has an history, it seem the same thing but we will distinguish it from the context. The idea is that the arrival sequence maps the arrival sequence in an history of operations.
Implemented it online with two approaches:

- Pessimistic
  - Based on locks, i.e., resource access control. Prevents the worst case scenario but putting a toll on the system(requiring waiting the end of an operation, risk of deadlock)
  - If a resource is taken, make the requester wait or pre-empt the holder
- Optimistic: implies much less waiting and killing of transaction. based on "age" of transaction and on it decide if holding or killing the transaction.
  - Based on timestamps and versions
  - Serve as many requests as possible, possibly using out-of-date versions of the data
  Commercial systems take the best of both worlds.

Locking

Approach traditionally most common. The idea is to have a lock of different type:

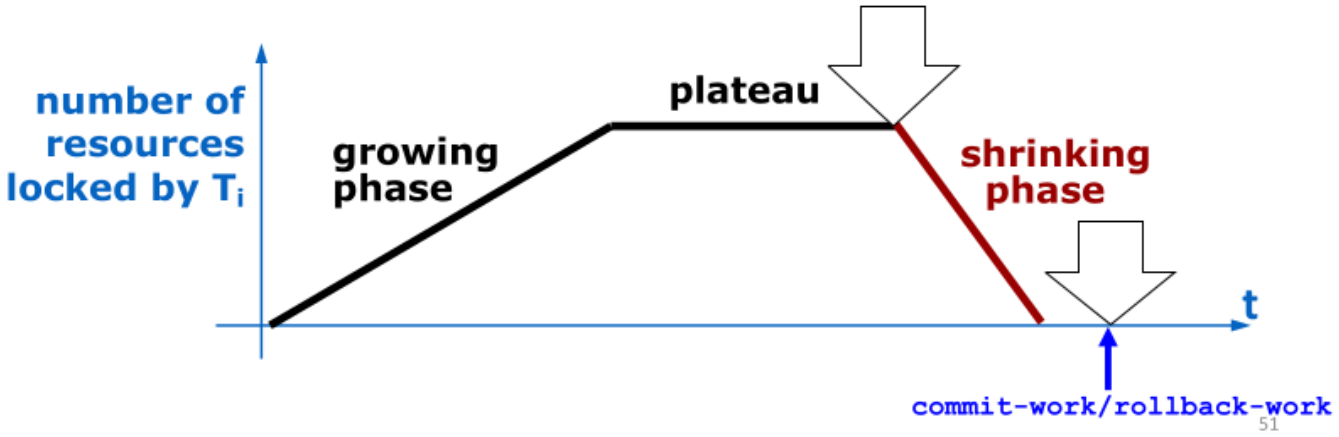
- Read lock or r_lock: we only want to read a resource, it not block the reading of the resource. It is a shared lock. At the end of the operation I have to unlock the resource.
- Write lock or w_lock: we acquire a resource to write on it. It is an exclusive lock as I don't want anybody to access the resource while I am using it. At the end I have to unlock the resource
  The access to exclusively locked resource usually is putted on hold until the resource is free. The arrival sequence will not coincide with the history as the locks impose some operations to be delayed.
  If a transaction want to read and then write and object it can acquire a w_lock already when reading or acquire a r_lock and upgrade it to a w_lock(naturally if it isn't shared with someone else).
  State of an object:
- free
- r-locked(locked by one or more reader)
- w-locked(locked by a writer)
  The lock manager receives requests from the transactions and grants resources according to the conflict table(in the table is assumed the request is done by a different transaction, if the lock request is granted the resource is acquired and when a lock is executed the resource become available):

Request	Resource status		
	FREE	R_LOCKED	W_LOCKED
r_lock	OK R_LOCKED	OK R_LOCKED (n++)	NO W_LOCKED
w_lock	OK W_LOCKED	NO R_LOCKED	NO W_LOCKED
unlock	ERROR	OK DEPENDS (n--)	OK FREE

How are locks implemented

Locks are applied on top of the physical implementation of DB and we use indices to represent data as they are really easy to use to represent data. You can use a key to find the correspondence between a key and its data. Constant time accesses. Sometimes we also use tree structure(B-tree) that have a hierarchical data structure, they are very good in organizing data regarding some attributes of the data. If we organize data in this way the lock is similar to a traffic light that permit/not permit to access data in lower nodes.
We can decide/imagine different ways to put locks sequence. Depending on these choice the sequence of action done can change. Just working with locks don't give guarantees to the correct execution of the transactions.
Two-Phase Locking (2PL)
Requirement (two-phase rule):

- A transaction cannot acquire any other lock after releasing a lock
- Sufficient to prevent non-repeatable reads (and more), serializability!



Meanwhile I acquire resources I am in the growing phase. At a certain point I stop and I go in a plateau phase and at the end I start releasing the resources in the shrinking phase until I don't have any more resources to free. Some anomalies cannot happen if I comply to this sequence(e.g non-repeatable read). The idea is that the scheduler decide if it can comply to the Two Phase Locking(2PL) it means it is included in the CSR class(it has stricter requirements).

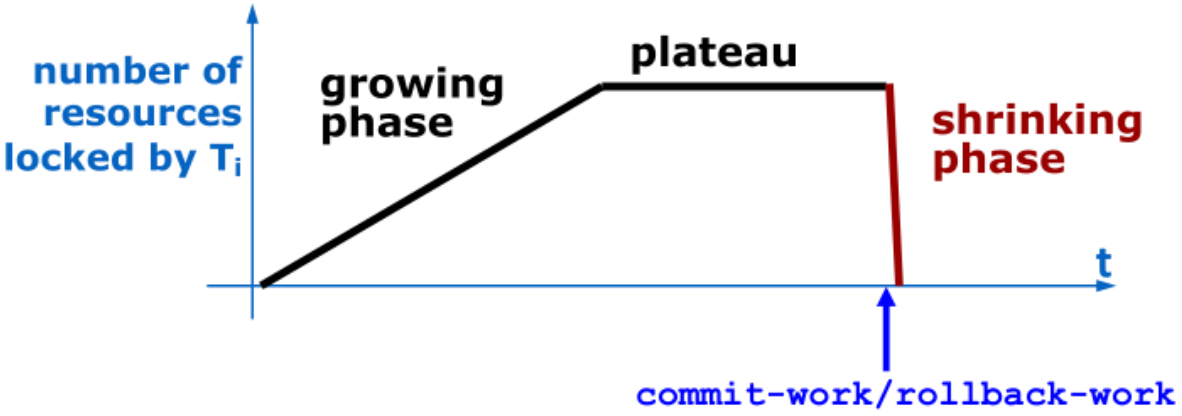
Assessing 2PL membership

The scheduler normally receives an arrival sequence, with also lock and unlock requests
The arrival sequence is transformed into a (possibly different) schedule (a-posteriori sequence)
When checking 2PL membership, we are focusing on the a-posteriori sequence, without the lock and unlock request
So when we wonder whether a “schedule” is 2PL, we ask:does it correspond to an arrival sequence to which we can add lock and unlock requests compatible with 2PL that would leave the sequence unchanged (without making any transaction wait)?
Under this interpretation, we again do not distinguish between the arrival sequence and the a-posteriori sequence
I can use resources visualization tables to see if the sequence of transactions is 2PL compliant.

Strict 2PL

Up to now, we were still using the hypothesis of commit-projection (no transactions in the schedule abort).
To remove this hypothesis, we need to add a constraint to 2PL, that defines strict 2PL:

- Locks held by a transaction can be released only after commit/rollback
- Remember: rollback restores the state prior to the aborted updates



Strict 2PL locks are also called long duration locks, 2PL locks short duration locks
Note: real systems may apply 2PL policies differently to read and write locks
Typically: long duration strict 2PL write locks, variable policies for read locks
NOTE: long duration read locks are costly in terms of performances: real systems replace them with more complex mechanisms

How to prevent phantom inserts: predicate locks

A phantom insertion occurs when a transaction adds items to a dataset previously read by another transaction
To prevent phantom inserts a lock should be placed also on “future data”, i.e., inserted data that would satisfy a previous query
Predicate locks extend the notion of data locks to “future data”
A transaction T = update Tab set B=1 where A<1
Then, the lock is on predicate A<1

- Other transactions cannot insert, delete, or update any tuple satisfying this predicate
  In the worst case (predicate locks not supported):
- The lock extends to the entire table
  In case the implementation supports predicate locks:
- The lock is managed with the help of indexes (gap lock)

Isolation Levels in SQL

SQL defines transaction isolation levels which specify the anomalies that should be prevented by running at that level
The level does not affect write locks. A transaction should always get an exclusive lock on any data it modifies, and hold it until completion (strict 2PL on write locks), regardless of the isolation level
For read operations, levels define the degree of protection from the effects of modifications made by other transactions
We use read-uncommit level of isolation when all my transaction only have to read.
We use read-commit to prevent dirty reads, but allows non-repeatable reads and phantom updates/inserts, some level of anomaly prevention, real systems can do more than the standard.
REPEATABLE READ avoids dirty reads, non-repeatable reads and phantom updates, but allows phantom inserts
SERIALIZABLE avoids all anomalies

	READ LOCKS	WRITE LOCKS
READ UNCOMMITTED	Not required	Well formed writes Long duration write locks
READ COMMITTED	Well formed reads Short duration read locks (data and predicate)	Well formed writes Long duration write locks
REPEATABLE READ	Well formed reads Long duration data read locks Short duration predicate read locks	Well formed writes Long duration write locks
SERIALIZABLE	Well formed reads Long duration read locks (predicate and data)	Well formed writes Long duration write locks

Serializable transactions don't necessarily execute serially
The requirement is that transactions can only commit if the result would be as if they had executed serially in any order
The locking requirements to meet this guarantee can frequently lead to a deadlock where one of the transactions needs to be rolled back

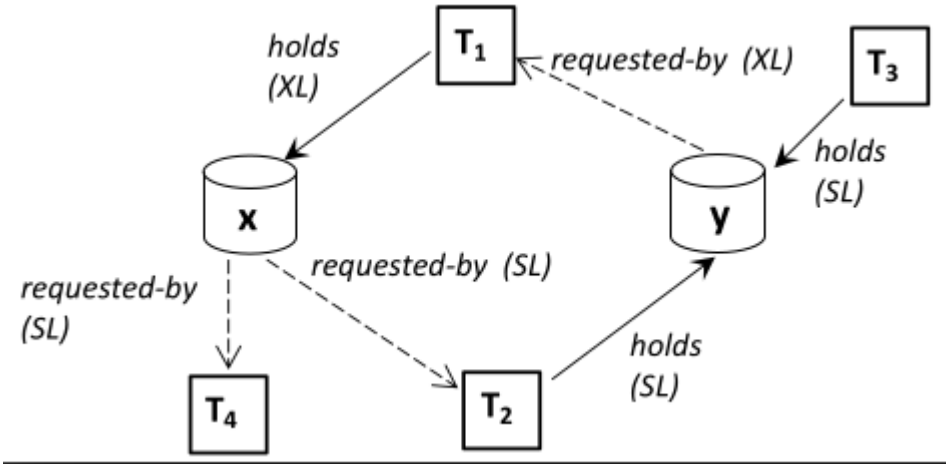
Therefore, the SERIALIZABLE isolation level is used sparingly and is NOT the default in most commercial systems

We need to pay attention on the locking waiting as it can create some infinite waiting time as two transactions want to write on a resource but they are waiting each others and cannot complete. For this in SQL there is a specification to add in the command to try to avoid these situations. We need to also pay attention to starvation as some transactions might never occur if they need to wait the right moment to be executed.

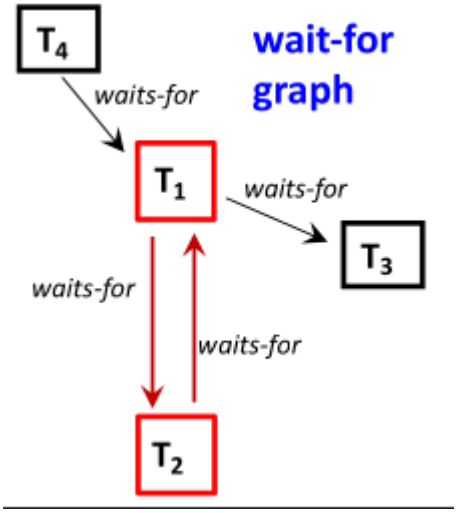
Deadlocks

Occurs because concurrent transactions hold and in turn request resources held by other transactions.

Lock graph: a bipartite graph in which nodes are resources or transactions and arcs are lock requests or lock assignments



Wait-for graph: a graph in which nodes are transactions and arcs are “waits for” relationships



We can also project a way in which we only take note of the transaction and take note in the graph only the transactions that are waiting for resources.

To destroy the cycle the simplest way is to kill a transaction(delete all its work and do it later hoping that the other transaction has finished).

Deadlock Resolution Techniques

Timeout: Transactions killed after a long wait
Deadlock prevention: Transactions killed when they COULD BE in a deadlock(Heuristics). Done either with Resources-based-prevention(restriction on lock request as transaction request all the resources all at once and only once, resources are globally sorted and must be requested in global order. The problem is that is difficult to anticipate all the requests) and Transaction-based prevention(restrictions based on transactions' ID. Assign an incrementing ID, we prevent older transactions from waiting for younger transactions to end their work. The choice on the transaction to kill can be Preemptive or Non-preemptive)
Deadlock detection: Transactions killed when they ARE in a deadlock(Inspection of the wait-for graph). Require an algorithm that detects the cycles in the wait for graph and it must work on distributed resources efficiently and reliably.

Deadlocks in practice

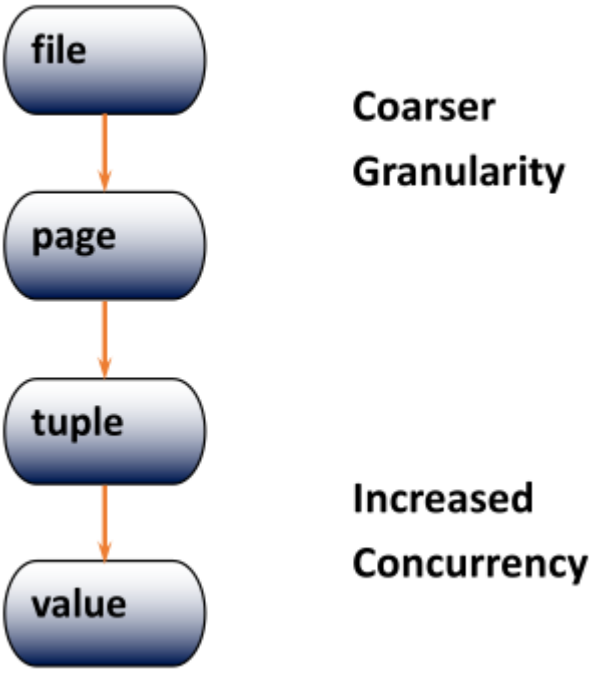
Their probability is much less than the conflict probability
Still, they do occur (once every minute in a mid-size bank). The probability is linear in the number of transactions, quadratic in their length (measured by the number of lock requests). Shorter transactions are healthier (ceteris paribus). There are techniques to limit the frequency of deadlocks

Update lock

I can tell the system that it is well behaved enough to have the resource be locked in a good way.
The most frequent deadlock occurs when 2 concurrent transactions start by reading the same resources (SL) and then decide to write and try to upgrade their lock to XL. To avoid this situation, systems offer the UPDATE LOCK (UL) asked by transactions that will read and then write
Update locks are easy to implement and mitigate the most frequent cause of collision: $r_1(x)$ $r_2(x)$ $w_1(x)$ $w_2(x)$. They are requested by using SQL SELECT FOR UPDATE statement

	Resource status			
Request	free	SL	UL	XL
SL	OK	OK	**OK**	No
UL	OK	**OK**	**No**	No
XL	OK	No	No	No

Hierarchical Locking



Locks can be specified with different granularities to permit multiple transaction to work on similar resource but on different levels. We want to lock the minimum amount of data and recognize conflicts as soon as possible. Method: asking locks on hierarchical resouces by requesting resources top-down until the right level is obtained and releasing locks bottom-up.
5 Lock modes:

- In addition to read (SHARED) locks (SL) and write (EXCLUSIVE) locks (XL)
  The new modes express the “intention” of locking at lower (finer) levels of granularity
- ISL: Intention of locking a subelement of the current element in shared mode
- IXL: Intention of locking a subelement of the current element in exclusive mode
- SIXL: Lock of the element in shared mode with intention of locking a subelement in exclusive mode (SL+IXL)
  Locks are requested starting from the root (e.g., starting from the whole table) and going down in the hierarchy.
  Locks are released starting from the leaves and going up in the hierarchy.
  To request an SL or ISL lock on a non-root element, a transaction must hold an equally or more restrictive lock (ISL or IXL) on its “parent”
  To request an IXL, XL or SIXL lock on a non-root element, a transaction must hold an equally or more restrictive lock (SIXL or IXL) on its “parent”
  When a lock is requested on a resource, the lock manager decides based on the rules specified in the hierarchical lock granting table

Request	Resource state: free	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	OK	No
IXL	OK	OK	OK	No	No	No
SL	OK	OK	No	OK	No	No
SIXL	OK	OK	No	No	No	No
XL	OK	No	No	No	No	No

Concurrent Control Based on Timestamps

A timestamp is the indication of the "age" of the transaction so the smaller the number the older the transaction. We want to prevent older transaction to modify value modified by younger transactions. This can also avoid some conflict that cause anomalies.
Locking is also named pessimistic concurrency control because it assumes that collisions (transactions reading-writing the same object concurrently) will arise. Alternative and complementary to 2PL (and to locking in general) are optimistic concurrency control methods.
In a distributed system you assign at each transaction the current time as its timestamp BUT you need to know there isn't a global time so there need to be an algorithm to assign unique timestamp to transactions(See Lamport clock).

TS concurrency control principles

The scheduler has two counters: RTM(x) and WTM(x) for each object
The scheduler receives read/write requests tagged with the timestamp of the requesting transaction:

- $r_{ts}(x)$:
  - If ts < WTM(x) the request is rejected and the transaction is killed
  - Else, access is granted and we set RTM(x) = max(RTM(x), ts)
- $w_{ts}(x)$:
  - If ts < RTM(x) or ts < WTM(x) the request is rejected and the transaction is killed
  - Else, access is granted and we set WTM(x) = ts

  Many transactions are killed

TS and Dirty Read

Basic TS-based control considers only committed transactions in the schedule, aborted transactions are not considered (commit-projection hypothesis). If aborts occur, dirty reads may happen.
To cope with dirty reads, a variant of basic TS must be used:

- A transaction T_i that issues a $r_{ts}(x)$ or $w_{ts}(x)$ such that ts > WTM(x) (i.e., acceptable) has its read or write operation delayed until the transaction T' that wrote the value of x has committed or aborted
- Similar to long duration write locks
- But…buffering operations introduces delays

2PL vs. TS

In 2PL transactions can be actively waiting. In TS they are killed and restarted
The serialization order with 2PL is imposed by conflicts, while in TS it is imposed by the timestamps
The necessity of waiting for commit of transactions causes long delays in strict 2PL
2PL can cause deadlocks, TS can be used to prevent deadlocks with the wound-wait and wait-die schemes explained before
Restarting a transaction costs more than waiting: 2PL wins!
Commercial systems implement a mix of optimistic and pessimistic concurrency control (e.g., Strict 2PL or 2PL + Multi Version TS)

Reducing kill rate: Thomas Rule

wts(x):

- if ts < RTM(x) the request is rejected and the transaction is killed
- else, if ts < WTM(x) then our write is "obsolete": it can be skipped
- else, access is granted and we set WTM(x) = ts
- Rationale: skipping a write on an object that has already been written by a younger transaction, without killing the transaction
  Works only if the transaction issues a write without requiring a previous read on the object (so SET X = X+1 would fail)

Multiversion Concurrency Control

The idea is for every resources we allow multiple transactions to write and we keep track of who wrote. In a point of time we have multiple version of a resource. We keep a single value for the reading transaction and it is the youngest transaction.
1^{st} **version(theory): TS-Multi allowing unordered writes**
Mechanism:

- $r_{ts}(x)$ is always accepted. A copy x_k is selected for reading such that:
  - If $ts \geq WTM_N(x)$, then k = N
  - Else take k such that $WTM_k(x) \leq ts < WTM_k + 1(x)$
- wts(x):
  - If ts < RTM(x) the request is rejected
  - Else a new version is created for timestamp ts (N is incremented)

- $WTM_1(x), \ldots, WTM_N(x)$ are the new versions, kept sorted from oldest to youngest
NB: this version shows what can be done in theory but is not the one used in the exercises
2nd version (practice):TS-Multi under Snapshot Isolation
Mechanism:

- $r_{ts}(x)$ is always accepted. A copy x_k is selected for reading such that:
  - If $ts \geq WTM_N(x)$, then k = N
  - Else take k such that $WTM_k(x) \leq ts < WTM_k + 1(x)$
- $w_{ts}(x)$:
- If ts < RTM(x) or $ts < WTM_N(x)$ the request is rejected
- Else a new version is created for timestamp ts (N is incremented)
- $WTM_1(x), \ldots, WTM_N(x)$ are the new versions, kept sorted from oldest to youngest
NB: this version is used in real systems based, e.g., on snapshot isolation (see later) and in the exercises

Snapshot Isolation (SI)

Give transaction some level of isolation as I made possible for transaction to read values related to the timestamp of the transaction. All the rights are of the transaction and it can work smoothly. When it finish there is a scheduler that decide if the transaction can commit or it has to be aborted/killed, this is done if its writes are in conflict with the writes of other concurrent transactions after the snapshot timestamp.

Anomalies in Snapshot Isolation

Snapshot isolation does not guarantee serializability

- T_1: update Balls set Color=White where Color=Black
- T_2: update Balls set Color=Black where Color=White
Serializable executions of T_1 and T_2 will produce a final configuration with balls that are either all white or all black
An execution under Snapshot Isolation in which the two transactions start with the same snapshot will just swap the two colors
This anomaly is called write skew