Concurrency control

Transactions per second (tps) is the measure of the workload of a operational DBMS; if two transactions access concurrently to the same data there is a problem: the module who resolve conflict avoiding interferences is *concurrency control*. He operates in order to increas DBMS efficiency in two ways:

. maximizing the number of transaction per second;

. minimizing the response time.

Operations can cause concurrency problems are:

. reading a data object $x \quad \Longrightarrow \quad r(x)$;

. writing a data object $x \quad \Longrightarrow \quad w(x)$.

It means that this operations may cause the reading on disk if the object is not present into the buffer or writing to disk an entire page of memory if, again, the object is going to be written does not fill in main memory.

The *scheduler* is a block of the concurrency control module who decide if an operation requested (read or write) can be satisfied; if it is not present is possible to have problems due to correctness, also called *anomalies*.

Concurrency problems

Examples of problems due to concurrency are:

. lost update;

. dirty read;

. inconsistent read:

  . ghost update (a);
  . ghost update (b).

Lost update

This effect can be shown when two transactions read the same object x and update concurrently its value: it happends that operation done by one transaction is lost.

Dirty read

This effect happends when a transaction reads the value of an object x who is not stable because the other transaction is still running so, for atomicity property, there will be a rollback cascade.

Inconsistent read

When a transaction reads the object x twice and x has different values the problem is called inconsistent read. It happends because between the two reads another transaction has modified the value of x. There are two kinds of inconsistent read:

. ghost update (a) if two transaction access concurrently to the same object and they view their modification each other; notice that all objects are already present into the database;

. ghost update (b) if one of two transaction insert a new object into the database and another transaction access use that data.

Theory of concurrency problem

Definitions of transactions and schedule operations are different:

. a transaction is a sequence of read/write operations characterized by the same transaction identifier (TID):

$$r_1(x)\, r_1(y)\, w_1(x)\, w_1(y)$$

. the schedule is a sequence of read/write operations presented by concurrent transactions (they appear in the arrival order of requests):

$$r_1(z)\, r_2(z)\, w_1(y)\, w_2(z)$$

The scheduler compiles schedules which can be accepted or refused by concurrency control in order to avoid anomalies. The scheduler scan and processed the order of execution without knowing what will be the results and, if it refuse a schedule, the transaction will be aborted.

Commit projection

This approach runs assuming that the schedule only contains transactions performing commit and cover all anomalies a part from the dirty read.

The most important concept introduced is *serial schedule*. A serial schedule is the schedule in which all actions of each transaction appear in sequence. For example:

$$r_1(x)\, w_1(x)\, w_2(x)\, r_2(x)$$

A *serializable schedule* is the schedule which produce the same results of an arbitrary serial schedule. There are some classes for equivalence between two schedules:

. view equivalence;

. conflict equivalence;

. 2 phase locking;

. timestamp equivalence.

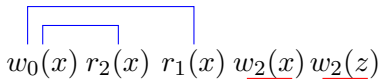
A single class defines how characteristic of schedules must be and how much is complex define the equivalence principle.

View equivalence In order to be equivalence, two schedule have to present the same set of *read-from* and *final-write*. A *read-from* means that the same object x was already written by another transaction, different from the one which is now reading x. A *final-write* means that the object x is written and there are no others transactions in the schedule that afterwards will write the same object.

If a schedule is *view serializable* (VSR) it means that is *view equivalent* to an arbitrary serial schedule which have the same transactions. For example, in the sequence:

$$w_0(x)\, r_2(x)\, r_1(x)\, w_2(x)\, w_2(z)$$

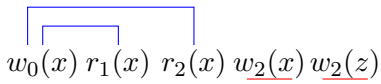
it is possible to find the following reletions:



where in blue are shown the *read-from* and in red the *final-write*. The sequence is serializable because it is equivalent to the following:

$$w_0(x)\, r_1(x)\, r_2(x)\, w_2(x)\, w_2(z)$$

Infact:



the two sequence present the same *read-from* and *final-write*.

View equivalence cover:

. lost update anomaly;

. inconsistent read anomaly;

. ghost update (a) anomaly.

To detect view equivalence has linear complexity if the schedule is given but detecting the equivalence to an arbitrary schedule is a NP problem (exponential complexity) so is not applied in real systems.

Conflict equivalence There is conflict if two actions operate on the same object ad at least one of them is a write operation. Conflicts can be:

. read-write (RW or WR: RW is similar to *final-write* operation, instead WR is similar to *read-from*);

. write-write (WW).

It is not possible have read-read conflicts otherwise the definition will not be respected.

Two schedules are *conflict equivalent* if they have the same conflict set and each conflict pair is in the same order in both schedules. Conflict equivalence is stronger than view equivalence because the schedule is strongly characterized in the first case.

If it is equivalent to an arbitrary serial schedule with the same transactions, a schedule is *conflict serializable* (CSR). For example the sequence:

$$w_0(x)\, r_1(x)\, w_0(z)\, r_1(z)\, r_2(x)\, r_3(z)\, w_3(z)\, w_1(x)$$

Conflict are:

. WR kind: $w_0(x) - r_1(x)$, $w_0(x) - r_2(x)$, $w_0(z) - r_1(z)$, $w_0(z) - r_3(z)$;

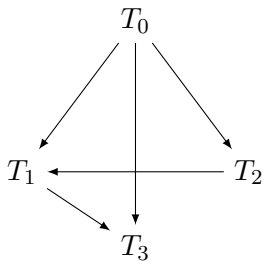
. RW kind: $r_1(z) - w_3(z)$, $r_2(x) - w_1(x)$;

. WW kind: $w_0(x) - w_1(x)$, $w_0(z) - w_3(z)$.

It is important notice that in the same transaction there are no conflicts. The conflicts discovered are the same of the serial sequence:

$$w_0(x)\, w_0(z)\, r_2(x)\, r_1(x)\, r_1(z)\, w_1(x)\, r_3(z)\, w_3(z)$$

In order to detect conflict it is possible to exploit the *conflict graph* avoiding the comparison with all possible serial sequecens; if the graph is acyclic the schedule is conflict serializable. This approach reduce complexity because checking graph cyclicity is linear in the size of the graph. For the sequence used in the example the graph will be the following:



The graph generates also constraints in the order of the optimal serial sequence because T_0 always precedes T_1, T_2, T_3 and so on; the full list of constraints is:

$$\begin{cases} T_0 \hookleftarrow T_1, T_2, T_3 \\ T_2 \hookleftarrow T_1 \\ T_1 \hookleftarrow T_3 \end{cases}$$

where the symbol $\hookleftarrow$ means: *precede*. Looking the list of the terms on the left from the top to the bottom it is possible to see the proper sequence for the serial one:

$$T_0, T_2, T_1, T_3$$

which is the one already reported before.

Although this method is less complex than *view equivalence* but also this approach is not used in real DBMS because have several disadvantages:

. possibility that the graph is large, so the evaluation is a long operation;

. the graph should be updated;

. the cycle absence operation have to be performed.

2 Phase Locking

This approach is the one really used by DBMS and it is based on *lock* operations. A *lock* the operation who block a specify resource in order to prevent any access by other transactions; it can be:

. a *read-lock* (R-lock);

. a *write-lock* (W-lock).

When the resource have to become free because it is not used more the system performs an *unlock* operation.

Each read operation have to be preceded by a R-lock request and followed by an unlock request; the same procedure is done for write operations.

The scheduler behaves like a lock manager: it receives transactions request and grants locks consequently. When the lock request is granted the corresponding resource is assigned to the requesting transaction and becomes unavaiable; only when the transaction require an unlock operation the resource can be accessed by others transactions. If the lock is not granted the requesting transaction is put in a waiting state which terminates only when the resource is unlocked: it allows locally reschedule policies.

The lock manager exploits the *lock table*, where is stored information about locks can be granted to a transaction and the *conflict table* which manage lock conflicts. The following table shows a conflict table:

Requests	Resource State		
	Free	R-Locked	W-Locked
R-Lock	Ok/R-Locked	Ok/R-Locked	No/W-Locked
W-Lock	Ok/W-Locked	No/R-Locked	No/W-Locked
Unlock	Error	Depends	Ok/Free

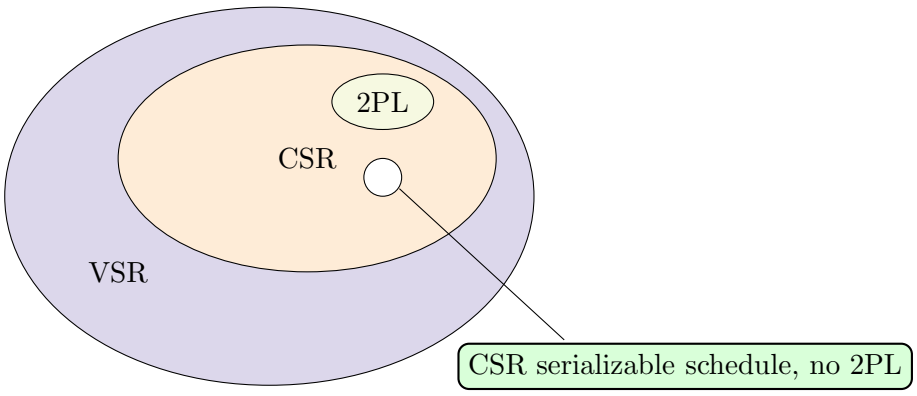
Read locks are shared: it means that other transaction may lock the same resources; each transaction holding the R-lock on the same resource is count with a counter: a resource is free when the counter is equal to 0.

2 Phase Locking is characterized by two phases:

. growing phase (lock are acquired);

. shrinking phase (lock are released).

The serializability is guaranteed because a transaction can not acquire a new lock after having released any lock. This method is strogest than conflict equivalence or view equivalence: the following picture shows levels of selectivity.



Strict 2 Phase Locking This method allows to drop the hypotesis done as far that only committed transaction were considered. In this case a transaction locks it is released only at hte end of the transaction (commit or rollback is not important): in this way data is always stable (dirty read avoided).

Techniques to manage locking When a transaction requests a resource x the system check if that resource is avaiable; in positive case the lock manager modifies the stat of x in its internal tables and returns the control to the transaction. In this approach the processing delay is very small. Otherwise if the resource is not avaiable the transaction is suspended and insered in the waiting queue; as soon as the resource becomes avaiable the first transaction present in the queue is resumed and processed. The probability of having conflicts is:

$$\mathcal{P}\left(Conflicts\right) \approx \frac{\kappa \cdot M}{N}$$

where:

- κ is the number of active transactions;
- M is the average number of objects accessed by a transactions;
- N is the number of objects present in the database.

If a timeout expire while a transaction is still waiting in the queue is possible that the lock manager extract or resume it or return an error code; the transaction, instead, can performs rollback and maybe restarts automatically or requests again the same lock after some time is passed.

Hierarchical Locking

In this approach locks can be acquired ad different levels:

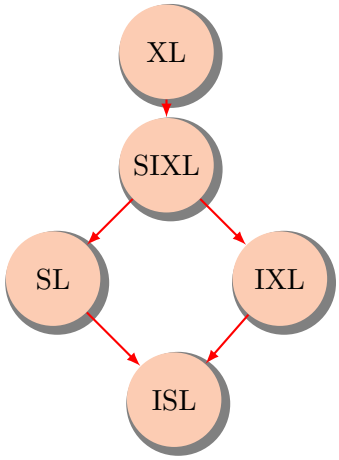
- table;
- group of tuples:
  - physical criteria (physical data page of memory);
  - logical criteria (tuples satisfating a given property);
- single tuple;
- single field of a tuple.

Hierarchical locking allows lock operations only if the transaction belong to the proper level and is characterized by a large set of locking primitives:

- shared lock (SL);
- exclusive lock (XL): allows both read/write operations;
- intention of shared lock (ISL): shows the intention of shared locking on a low hierarchical level;
- intention of exclusive lock (IXL): shows the intention of exclusive locking on a low hierarchical level;

. shared lock and intention of exclusive lock (SIXL): shared lock of the current object and intention of exclusive locking on a low hierarchical level of one or more objects.

In the following picture is shown the hierarchy:



Locks are always requested starting from the root of the tree and going down: this is also the way followed in order to perform *localized* reads (detailed granularity). Another option is reading from the leaves node to the root: *massive* reads are performed (rough granularity). Both approaches allow to reduce concurrency, but can create overhead if granularity is too fine.

Until now, all anomalies are covered with methods studied, a part from ghost update type b (update performed by a transaction with insertion of data by another transaction). Infact, 2PL allows locks only for objects already present into the database. *Predicate locking* admits lock for all data satisfating a given predicate: the real implementation is made through *locking indices* who prevent the insertion of new tuples.

Transactions are divided into two classes:

. read-write (default);

. read only: it not allows to perform updates.

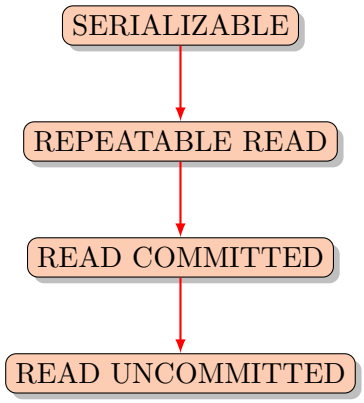
The way in which transactions interacts each others is specify by the *isolation level*. There are four levels:

. **serializable**: is the highest level, implemented by strict 2PL and includes predicate locking, so cover all anomalies;

. **repeateble read**: implemented by strict 2PL without predicate locking cover all anomalies a part from ghost update type b;

. **read committed**: does not implement 2PL and release the lock when read is ultimate; it means that *dirty read* is avoided because write operation does not release the lock (remember that usually locks are released only when transaction ends);

. **read uncommitted**: data is read without locking; it is possible because read operations does not affected database, infact this level is allowed only for read-only transactions and, for this reason, *dirty read* is not avoided.



Since write operations are critical because can propagate errors through database, they only can be executed under strict 2PL with exclusive locking.

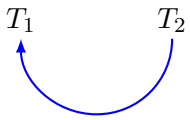
Deadlock

This is a typical situation for concurrent systems managed by locking and waiting times. It happends when one transaction (T_1) is waiting for a resource locked by another transaction (T_2) and (T_2) is waiting for a resource locked by (T_1).

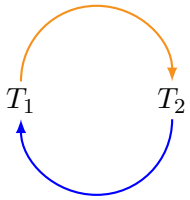
The easier possible solution is adopting *timeout*: after sometime, if the period specify by the timeout is expired, the transaction rollback (maybe can restart completly or re-asking another lock operation). If the timeout is too long it happends that the system waits for a long period; othewise is possible that too much transactions are killed.

In distributed database deadlock are detected by the *wait graph*. In the previous example, (T_1) was waiting for a resource reserved by (T_2); in the following picture is shown this condition:

But also (T_2) is waiting for a resource locked by (T_1):



So:



In this case there is a cycle which represents a deadlock. This approach is too expensive to build and manage.