Transaction Models and Concurrency Control

Alberto Abelló

PID_00184140

Universitat Oberta de Catalunya

Index

Introduction

This module of the subject *Database Architecture* will introduce you to advanced issues in concurrency control management. A transaction is the atomic execution unit of a database management system (DBMS), and concurrency control manager is the component of the DBMS in charge of providing the mechanisms guaranteeing that concurrent transactions generate the same results as if they were executed in a non-concurrent environment.

We will first recall some basic concepts you should remember about interferences, isolation levels and correctness criteria before we go further. Afterwards, we will explain some techniques that make locking mechanisms more flexible. Then, we will introduce timestamping as a more optimistic alternative to locking. Timestamping evolves into multi-version concurrency control, which is implemented in many DBMSs, like for example Oracle and PostgreSQL. We will analyze pros and cons of those alternatives, before discussing some transaction tuning issues (including transaction chopping).

The second part of the module will increase the complexity by going throughout concurrency control in distributed systems. First, we will see how locking and timestamping mechanisms can be adapted to these environments and the difficulties this raises. Finally, we will also study different alternatives to deal with replica management, underlining the importance of this in today highly distributed database systems.

Objectives

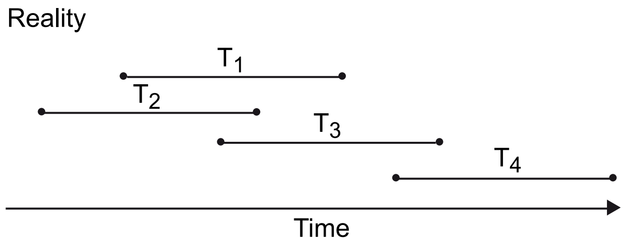
The main objective of this module is to present advanced concepts of concurrency control. Specifically:

1. Remember the four kinds of interferences.
2. Remember the correctness criteria for concurrent access scheduling.
3. Analyze the benefits and drawbacks of some concurrency control mechanisms: locking, timestamping and multiversion.
4. Know some tuning mechanisms for locks.
5. Understand the concurrency control mechanisms of Oracle 10g.
6. Explain the difficulties of distributed concurrency control.
7. Understand the difficulties of locking in a distributed environment.
8. Understand the assignment of timestamps in a distributed environment.
9. Identify the difficulties in highly distributed systems: avoiding interferences, avoiding deadlock situations and dealing with replicas.

1. Interferences and Isolation Level

Nowadays, most (if not all) information systems have a requirement for concurrent access of users. This means that many users can access their data at the same time.

Figure 1. Concurrent transactions



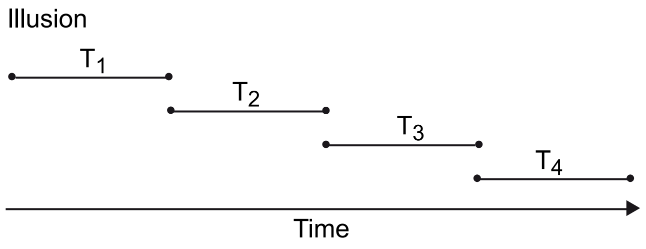
Thus, Figure 1 sketches how the transactions (i.e., T_1, T_2, T_3 and T_4) issued by different users overlap. This overlapping may result in what one user does interfering with other users.

Transaction

"A transaction is a tool for application programmers to delegate the responsibility for preventing damage to data from threats such as concurrent execution, partial execution, or system crashes to the database system software."

Gottfried Vossen, Encyclopedia of Database Systems

Figure 2. Illusion of isolation



Any user must obtain the very same results in the presence of other users as if he or she was working alone. Thus, it is the responsibility of the DBMS to generate the illusion that users work sequentially (sketched in Figure 2). More specifically, the concurrency control manager is in charge of this task.

> "The concurrency control manager synchronizes the concurrent access of database transactions to shared objects in the database. It will protect each transaction from anomalies (i.e., Lost update, Read uncommitted, Unrepeatable read, and Inconsistent analysis) that can result from the fact that other transactions are accessing the same data at the same time."
>
> Andreas Reuter, Encyclopedia of Database Systems

In previous courses, you studied the basics of this component of the DBMS. Before going throughout advanced concurrency control issues, let us briefly remember some concepts and definitions.

1.1. Schedules and Serializability

The sequence of steps in the execution of a set of transactions is called a schedule. It contains operations over granules. We will consider that before writing any granule, the DBMS always needs to read it. Thus, in this module, we will assume the scheduler gets three different kinds of operations from the query optimizer, namely read (R), read for update (RU) and write (W).The first one is generated by the transaction manager on finding a `SELECT` statement, while the other two are generated on getting `INSERT`, `UPDATE` or `DELETE` statements.

Figure 3. Example of schedule

#time	T_1	T_2
1	R(A)	
2		RU(A)
3	R(B)	
4		W(A)
5	R(C)	
6	commit	
7		commit

$RS(T_1)=\{A,B,C\}$
$WS(T_1)=\emptyset$
$RS(T_2)=\{A\}$
$WS(T_2)=\{A\}$

You can see an example of schedule in Figure 3. Here, two transactions operate over granules[1] *A*, *B*, and *C*. For each one of the transactions we can record the set of granules read (i.e.,ReadSet -RS) and the set of granules written (i.e.,WriteSet -WS). In the example, the read set of T_1 and T_2 is respectively $\{A, B, C\}$ and $\{A\}$. In the same way, the write set of T_1 and T_2 is respectively $\emptyset$ and $\{A\}$.

(1)We will use this term to refer to the unit of data the DBMS is accessing, either records, blocks, files, etc.

Note

You should notice that $WS(T) \subseteq RS(T)$, since we always read a granule before writing it.

Figure 4. Example of serial schedule

#time	T_1	T_2
1	R(A)	
2	R(B)	
3	R(C)	
4	commit	
5		RU(A)
6		W(A)
7		commit

$RS(T_1)=\{A,B,C\}$
$WS(T_1)=\emptyset$
$RS(T_2)=\{A\}$
$WS(T_2)=\{A\}$

Remember also that a *serial schedule* (like that in Figure 4) corresponds to a noninterleaving execution of the transactions (i.e., the execution intervals of the transactions, from the first operation to the commit, do not overlap). A schedule whose effect is equivalent to that of some serial execution of the same set of transactions is called serializable. Serializability theory concerns finding properties of schedules that are serializable. Remember that this theory assumes that transactions always end with commit. Thus, if not said otherwise, throughout this module we will assume that the system is reliable (i.e., no power or hardware failure can occur) and no transaction is canceled. Also until section 5, we will consider that data are not replicated.

"Recovery ensures that failures are masked to the users of transaction-based data management systems by providing automatic treatment for different kinds of failures, such as transaction failures, system failures, media failures and disasters."

Erhard Rahm, Encyclopedia of Database Systems

1.2. Kinds of Interferences

There are four kinds of interferences that can make a schedule not to be serializable:

- **Lost update** (also known as Write-Write) appears when the data written by a transaction is lost, because another one overwrites them before it commits. For example, T_1 reads the balance of a bank account and gets the value 100; then T_2 reads the same value from the same account; afterwards, T_1 adds 25 € and writes the value; and finally T_2 adds 50 € (to the value it read) and writes the value. The result of such execution is that the sum done by T_1 has been lost (i.e., the final balance is 150 € instead of 175 €).

- **Read uncommitted** (also known as Write-Read) typically appears when a transaction reads (and uses) the value written by another transaction and the one who wrote it, because of one reason or another, in the end, does not commit its results. For example, T_1 attempts to get some money from the cash machine; then T_2 (from another cash machine) reads the balance of the same account and sees that T_1 got the money; and finally T_1 cancels and does not get the money. The result is that T_2 saw a balance that never existed.

- **Unrepeatable read** (also known as Read-Write) appears when a transaction reads some data, another transaction overwrites those data, and the first one tries to read the same data again (it will find them changed). For example, T_1 reads the balance of an account (which is 50 €); then T_2 (from another cash machine) gets 25 € from the same account; finally, after considering the balance previously retrieved, T_1 tries to get 50 €, which is surprisingly not possible now.

- **Inconsistent analysis** (a special case of which are Phantoms) appears when the result of accessing several granules is affected by changes made by other transactions, so that it does neither reflect the state before, nor after the execution of those other transactions. For example, T_1 lists the names of passengers in a flight; T_2 inserts a new passenger in that flight; finally, T_1 counts the number of passengers in the flight. As a result of such concurrent execution we get a phantom, i.e., a passenger that is counted and whose name is not in the list. It is important to notice that the difference between unrepeatable read and this is that the former affects one granule, while this involves more than one granule (i.e., one tuple is read and a different one is written).

If transactions in a schedule do not interfere one another, then there is a serial schedule equivalent to it, and the former is considered serializable.

1.3. Locking Basics

Typically, relational DMBSs avoid those interferences by locking data. Before working on a granule *G*, a transaction *T* needs to lock it. Different locking modes exist depending on the operation *T* has to execute on *G*.

Figure 5.

	S	X
Shared	OK	NO
eXclusive	NO	NO

Compatibility matrix for Shared-Exclusive locking

The basic mechanism underlying all implementations is Shared-Exclusive locking, whose compatibility matrix is in Figure 5. We say that two locking modes are compatible if two transactions that access the same granule can obtain these locks on the granule at the same time.

See also

In section 2.1., we will see read-for-update locking and multi-granularity locking mechanisms, that use a different compatibility matrix.

What all locking techniques have in common is that for each granule they store the list of transactions locking it and the corresponding mode; also for each granule a queue of transactions (that follows a strict FIFO policy) waiting for it to be unlocked as well as the corresponding mode for these transactions; and a unique matrix of compatibility between locking modes.

First In First Out

A FIFO policy is used in the queues to avoid starvation (i.e., low priority transactions never acquiring the lock on the granule).

Besides those three components, two operations must be implemented:

1) *Lock (G,m)*, which locks granule *G* in mode *m* for *T*:

- If *G* is not locked by any other transaction, it is locked for *T* and *T* continues its execution.
- If *G* is locked by other transactions in a compatible mode, it is also locked for *T* and *T* continues its execution.

- If *G* is locked by other transactions and one of them locked it in a mode incompatible with mode *m*, *T* is blocked and queued until all incompatible locks are released.

2) *Unlock* (*G*), which releases the lock of *T* over *G*:

- If there is not any queued transaction waiting, nothing has to be done.
- If there are queued transactions, we follow the order in the queue to unlock as many transactions as possible according to the two situations bellow, until the first transaction in the queue cannot be dequeued:
  - If there is no other transaction holding the lock, the fist transaction in the queue takes the lock, is unblocked and dequeued.
  - If there are other transactions holding the lock, we have to check whether the mode of the first transaction in the queue is compatible with that of the current holders. If so, this also gets the lock and is unblocked and dequeued.

Ju st defining these two operations is not enough. A locking protocol is also necessary. All locking concurrency control mechanisms assume wellformedness of the sequence of locks generated by a transaction, which consists on:

1) A transaction will never execute an operation over *G* without getting the lock in the right mode.

2) A transaction can only unlock those granules previously locked by it.

3) A transaction cannot lock twice the same granule if the second mode is not stronger (i.e., more restrictive) than the first one. In order to lock a granule twice, the granule must be unlocked in between the two locks. An exception to this is that a lock over a granule can be reinforced (i.e. locked in a stronger mode) without unlocking it.

4) A transaction will eventually release all locks.

Transactions also have to follow the *two phase locking* (2PL) protocol. This protocol states that a transaction cannot request a lock after it releases another one. This clearly defines two phases in each transaction (namely growing and shrinking).

In order to guarantee serializability, a transaction must be wellformed and have two phases. During the first phase, locks are only acquired; and during the second one locks are only released.

Surprisingly, using locks and applying these simple rules guarantees serializability, but does not prevent all interferences. More specifically, we could still read uncommitted data, because it has to do with recoverability, which is overlooked by serializability theory. As mentioned before, this theory ignores the possibility of canceling a transaction.

Kinds of cancellation

We distinguish two different kinds of cancellations (namely *rollback* and *abort*) depending on whether they are directly induced by the user or by the concurrency control component of the DBMS, respectively.

Besides locking, we need to enforce also that no transaction T_2 that accessed a granule written by T_1 commits before T_1.

Wait-For graph

A Wait-For graph contains one vertex per transaction, and the directed edges show that the tail of the edge is queued waiting for the head to release the locking of some granule. Usually, the edge is tagged with the identifier of the granule.

To enforce this property, we need to apply what is known as *strict two phase locking* (aka, Strict 2PL). The strictness of the protocol comes from the fact that granules are typically locked and unlocked automatically by the system without the intervention of the user. Thus, it is impossible to determine when the user did the last lock (so that others can be released). The solution to this is to atomically unlock everything inside the commit operation.

Figure 6. Schedule with SX-locking

#time	T_1	T_2
1	lock(A,S)	
2	R(A)	
3		lock (A,X)
4	R(A)	🟩
5	commit (unlock (A))	🟩
6		RU(A)
7		W(A)
8		commit (unlock (A))

$$\begin{tikzcd} T_2 \arrow[r, "A"] & T_1 \end{tikzcd}$$

Figure 6 shows an example illustrating how SX-locking works. The first column contains a numbering that will help us to reference all operations. The other two columns contain the operations of one transaction each (i.e., T_1 and T_2 respectively). At time 1, T_1 locks granule A for reading (mode shared), just before performing the read operation over the granule. Afterwards, T_2 does exactly the same. However, the actual intention of T_2 is modifying the granule and it generates a lock operation over the same granule in exclusive mode to state its real purpose. Unfortunately, S and X modes are not compatible according to the matrix in Figure 5. Therefore, T_2 is queued waiting for T_1 to unlock the granule A. This is shown in the Wait-For graph at the right side of the figure. Afterwards, T_1 reads A again (note that it does not have to be locked before, because it still is), and commits. As explained before, the commit operation itself is in charge of unlocking everything, following the Strict-2PL. Once the lock is released, T_2 is dequeued and comes back to execution. It reads granule A and writes its new value. Finally, it commits, releasing its lock. It is important you realize that thanks to those locks we avoided an unrepeatable read interference. If T_2 had been allowed to write A before the second read of T_1, this had retrieved a value different from that retrieved by the first reading.

Therefore, by making T_2 to wait, we avoid having inconsistent results in T_1. Now this schedule is serializable, since its result is equivalent to the serial one executing firstly T_1 and T_2 afterwards.

1.4. Isolation Levels

Concurrency control is not always necessary. You should notice that to have interferences, you must be in a multiuser environment and at least one of those users must be writing something. Thus, in mono-user or read-only environments concurrency control is absolutely useless and a waste of resources. In general, between the two extreme situations (i.e., many potential interferences where seralizability is mandatory, and no interferences at all or they do not matter) we have many possibilities.

Data Warehousing

A typical example of deactivating concurrency control are data warehouses, since all their users are read-only.

The standard SQL defines four different isolation levels that a DBMS can guarantee (i.e., `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ` and `SERIALIZABLE`). Each one of these isolation levels avoids one more kind of interferences than the previous one. Standard SQL does not assume any implementation for isolation. Nevertheless, SX-locking clearly fits into the four standard isolation levels.

Setting the isolation level

According to standard SQL, the isolation level of a transaction can be set by means of the statement `SET ISOLATION LEVEL <levelName>;`.

Let us analyze how they can be implemented:

- `READ UNCOMMITTED`: All we need to avoid lost update interferences is exclusive locks before writing granules. These locks must be released at the end of the transaction. To avoid this kind of interferences (i.e., Write-Write), it is unnecessary that transactions lock granules just to read them.

- `READ COMMITTED`: Just locking for writing is not enough to avoid read uncommitted (i.e., Write-Read) interference. We need to check whether the granule we are going to read is in an uncommitted state or not. To do this, it is only necessary to check whether we can lock the granule in shared mode. We do not need to keep the lock until the end of the transaction, but just get the lock and immediately (at the end of the read operation) release it.

- `REPEATABLE READ`: In order to avoid unrepeatable read (i.e., Read-Write) interferences, we have to keep the shared lock until the end of the transaction (i.e., we have to follow the Strict-2PL). By keeping the lock, we guarantee that if we have to read a granule twice, its value will have not changed from the first reading to the second. Actually, it would be enough to keep the lock until the second read operation. However, since the concurrency control manager does not know whether there is going to be a second reading or not, we have to keep the granule locked until the commit command is received.

- `SERIALIZABLE`: Since inconsistent analysis interferences involve more than one granule, to be able to avoid them, the DBMS has to lock not only data, but also metadata in the catalog. Thus, in the example of phantoms we saw above, by locking the control information of the table, we would avoid the insertion of new tuples while we are scanning it.

In order to decide the most appropriate isolation level for a transaction, you should analyze the requirements of your application and decide which one suits your needs. In many applications, the `SERIALIZABLE` isolation level is not really necessary.

Having a phantom

Having a phantom in the list of passengers of a flight is clearly critical, while having a phantom on retrieving the average salary of a company with thousands of employees is probably not.

Figure 7. Percentage of correct results depending on the number of concurrent transactions

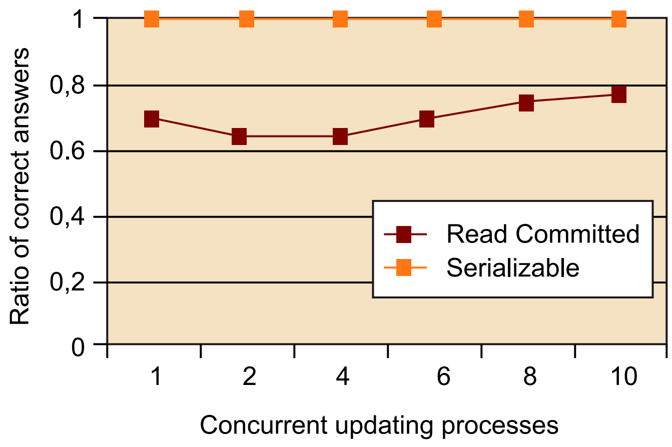
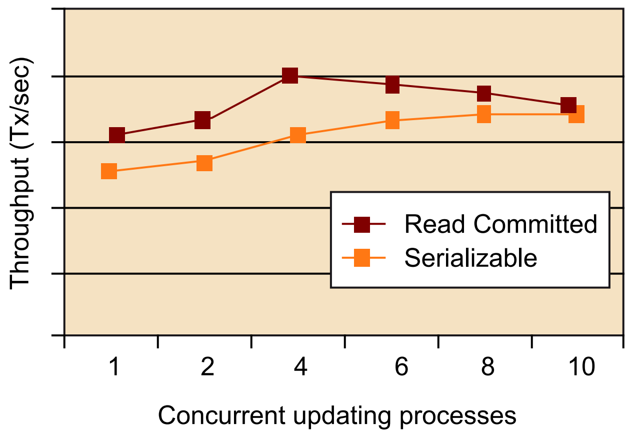


Figure 7 depicts the correlation between concurrent threads and interferences for two different isolation levels. Using `SERIALIZABLE` you guarantee that 100% of transactions are safe. However, using `READ COMMITTED` approximately 25% of the transaction will have some interferences. You should ask yourself whether these interferences can be assumed in the context of your application or not.

Figure 8. Throughput (transactions per second) depending on the number of concurrent transactions.



On the other hand, Figure 8 depicts the correlation between the number of concurrent threads and the throughput of the system (measured in terms of transactions per second). Clearly, when using `READ COMMITTED` isolation level you obtain higher throughput than using `SERIALIZABLE` (just because you do not keep shared locks until the end of the transactions).

Thus, the trade-off should be clear for you now: The more you lock the more you penalize performance; the less you lock the less you guarantee consistency of your transactions.

In general, how the isolation level affects consistency and performance depends on the size of the intersection of write and read sets of different transactions, and the time that those transactions keep the locks.

2. Centralized Concurrency Control

Concurrency control has to deal with the problems generated from many users accessing the same data. In this section we are going to consider the easy case of a centralized DBMS, where data are in one machine and users directly connect to it.

See also

In section 4 will explain the problems that arise when data is distributed along different locations and the entry point of users to the system can also be any of those locations.

2.1. Advanced Locking

SX-locking is powerful enough to guarantee serializability and flexible enough to implement the four isolation levels in the standard. However, it is too rigid to smoothly accommodate many real world situations. The problem of SX-locking is that, most of the times, it unnecessarily reduces concurrency, generates contention and incurs in deadlock situations, resulting in a diminution in the throughput of the whole system.

2.1.1. Read-for-update Locking

Figure 6 exemplified how read-for-update operation generates exclusive locking. You should notice that read-for-update operation is not really necessary (SX-locking also works with only read and write operations), but it just saves reinforcing the locking over granule *A*.

Reinforcing locks is a source of deadlock situations. We first lock in shared mode, and only at the moment of writing, the lock is promoted to exclusive mode. The problem is that another transaction can be doing the same at the same time. Thus, both keep the shared lock and try to get the exclusive one.

Figure 9. Schedule with SX-locking and lock reinforcement

#time	T_1	T_2
1	lock(A,S)	
2	RU(A)	
3		lock (A,S)
4		RU(A)
5	lock (A,X)	
6	🟩	lock (A,X)
7	🟩	🟩
8	🟩	🟩

$$\begin{tikzcd} T_2 \arrow[r, bend left, "A"] & T_1 \arrow[l, bend left, "A"] \end{tikzcd}$$

Figure 9 shows an example of this situation. First, T_1 locks *A* in mode shared, while T_2 does exactly the same. Then, T_1 tries to reinforce the lock as well as T_2. You can see the Wait-For graph at the right hand side of the figure. It

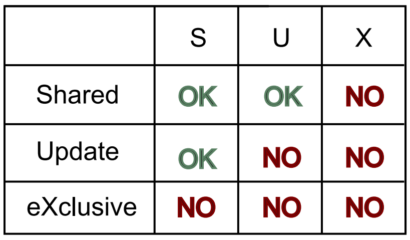
Note

A cycle in the Wait-For graph corresponds to a deadlock situation.

reflects that T_1 is waiting for T_2 to release the lock, and T_2 is also waiting for T_1 to release the lock of the very same granule, resulting in a typical deadlock situation.

Thus, thanks to directly generating an exclusive locking on getting a read-for-update, we avoid such deadlock situations. However, we reduce concurrency, because we use exclusive mode longer than strictly necessary. During the time elapsed between the read and write operations, the granule is unnecessarily locked in exclusive mode.

Figure 10.



	S	U	X
Shared	OK	OK	NO
Update	OK	NO	NO
eXclusive	NO	NO	NO

Compatibility matrix for Read-for-Updade locking

In order to avoid this undesired side effect, we have to define a more elaborated locking technique. As in the basic one, we have to keep for each granule the list of transactions locking it and the corresponding mode, and a queue of transactions waiting for it to be unlocked and also the corresponding mode for these. The difference is that we define a more complex compatibility matrix with a new mode *U* corresponding to the new read-for-update operation. This matrix is shown in Figure 10 and its semantics are as follows:

- Shared is used for reading the granule.
- Update is used for immediate reading and future writing.
- eXclusive is used for writing.

Together with this new matrix, the concurrency control manager also has to change the protocol to deal with the locks. On getting a lock for update, it creates a private copy of the granule that other transactions cannot see and the locking transaction works exclusively on it. Then, exclusive locking is only performed at commit time, and this operation is in charge of making public all the existing private copies.

Figure 11. Schedule with SUX-locking not generating contention

#time	T_1	T_2
1	lock(A,S)	
2	R(A)	
3		lock (A,U)
4		RU(A)
5		W(A)
6	R(A)	
7	commit (unlock (A))	
8		lock(A,X)
9		commit (unlock (A))

Figure 11 exemplifies how by means of this technique we avoid queuing T_2 (you should compare it with the example in Figure 6 where we use exclusive instead of read-for update locking). Now, at time 3, we try to lock granule A for update. At this point, a copy of this granule is created, so that T_1 keeps on working over the original, while T_2 works over the copy. In this way, both can work in parallel without interfering each other. It is at time 9 that the private copy of T_2 is made public. You should notice that only one copy can exist at the same time, because U mode is not compatible with other transaction locking the same granule exclusively or for update. Nevertheless, it is still necessary to reinforce the lock at commit time to wait for transactions that already read the public copy and did not finished, yet (they could try to repeat the reading and this second one should be consistent with the former). Finally, also at commit time T_2 makes public the private copy of A.

Figure 12. Schedule with SUX-locking generating contention

#time	T_1	T_2
1	lock(A,S)	
2	R(A)	
3		lock (A,U)
4		RU(A)
5		W(A)
6		lock (A,X)
7	R(A)	🟩
8	commit (unlock (A))	🟩
9		commit (unlock (A))

$$\begin{tikzcd} T_2 \arrow[r, "A"] & T_1 \end{tikzcd}$$

However, this technique does not completely avoid contention. First, two transactions trying to modify the same granule are not compatible, and one of them will be locked. Moreover, one read-only transaction can also force an updating one to wait for it to finish as depicted in Figure 12. At time 6, T_2 has to wait for T_1 to finish, in order to avoid an unrepeatable read interference.

The main advantage of this technique is that it avoids some deadlocks and allows more concurrency by means of the private copy of the granules. The disadvantage is that managing private copies incurs in some overhead.

2.1.2. Multi-granule Locking

One may think that locking the finest granularity is always the best possibility. However, common sense does not work in this case. There are some reasons not to choose this option (i.e., why locking coarser granularities is better).

First of all, locking coarser granules, we have to manage less contention and undo information. This results in a reduction of the overhead due to concurrency control. Moreover, imagine now that we have two transactions accessing several rows in the same table. If they lock the records one by one, there is some probability of having a deadlock. However, if they try to lock the whole file at once, only one of them will access it. In this situation, we loose concurrency (in the sense that less transactions can actually work in parallel), but we completely avoid deadlocks (at least those involving only rows in this table).

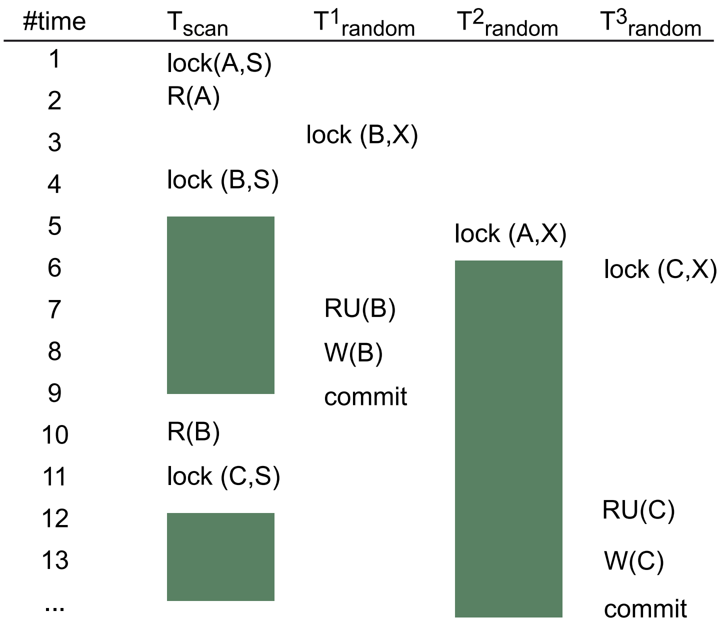
Undo information

This is the information stored in the log in order to be able to rollback transactions.

Note

For the sake of simplicity, from here on, we will omit unlocks in the figures and just assume they are all done inside the commit or cancel operation of each transaction.

Figure 13. Schedule of a full table scan with fine granularity



#time	T_{scan}	T^1_{random}	T^2_{random}	T^3_{random}
1	lock(A,S)			
2	R(A)			
3		lock (B,X)		
4	lock (B,S)			
5	🟩		lock (A,X)	
6	🟩		🟩	lock (C,X)
7	🟩	RU(B)	🟩	
8	🟩	W(B)	🟩	
9	🟩	commit	🟩	
10	R(B)		🟩	
11	lock (C,S)		🟩	
12	🟩		🟩	RU(C)
13	🟩		🟩	W(C)
...			🟩	commit

Another interesting scenario to pay attention to, depicted in Figure 13, is when we have many transactions (T^n_{random}) each one of them modifying randomly one tuple in a table and another one (T_{scan}) scanning the whole table. For the sake of simplicity, we assume *SX*- instead of *SUX*-locking (the effect would be similar for *SUX*). Let us suppose now that T_{scan} starts scanning the table. Sooner or later, it will try to access a record locked by T^1_{random} and will be queued. In the meanwhile, T^2_{random} will try to access a record locked by T_{scan} and will also be queued. Then, T^{3}_{random} locks a record that nobody was accessing. Finally, T^{1}_{random} commits, releases the lock and T_{scan} can continue reading but only until the record locked by T^{3}_{random} (notice that T^{2}_{random} is still queued). Before the end of T^{3}_{random}, other transactions could try to access this table resulting in either potentially locking T_{scan} and retarding its advance or being locked by it. Alternatively, as depicted in Figure 14, if T_{scan} locks the whole file from the beginning (instead of record by record), it will not find so many obstacles, because it will only be queued in those rows locked by transactions previous to it. Those transaction that begin after T_{scan} started the table scan and try to access rows in the table (either already accessed or not by T_{scan}) are automatically queued. Thus, they cannot retard the end of T_{scan}. Therefore the advantage is double since T_{scan} will take less time to finish, and consequently those T_{random} will also take less time queued waiting for its end. This is graphically reflected in the figures in that the area of dark rectangles is smaller in Figure 14 than it was in Figure 13.

Figure 14. Schedule of a full table scan with coarse granularity

#time	T_{scan}	T^{1}_{random}	T^{2}_{random}	T^{3}_{random}
1	lock(File,S)			
2	R(A)			
3	R(B)			
4	R(C)			
5		lock (B,X)		
6		🟩	lock (A,X)	
7		🟩	🟩	lock (C,X)
8	commit	🟩	🟩	🟩
9		RU(B)		
10		W(B)		
11		commit		
12			RU(A)	
13			W(A)	
14			commit	
15				RU(C)
16				W(C)
17				commit

It is important the DBMS allows the user to choose the locking granularity, instead of always locking records. However, choosing the right granularity is not an easy task for a database administrator, and managing multiple granularities is not easy either for the DBSM.

SQL statement to lock a table

```
LOCK TABLE <name> IN
[SHARE|EXCLUSIVE]
MODE;
```

As a rule of thumb, you should use coarse granularities (like files or tables) for long transaction accessing many data, and fine granularities (like records) for short transactions accessing only few data.

Figure 15.

	IS	IX	S	X	SIX
Intentional Shared	OK	OK	OK	NO	OK
Intentional eXclusive	OK	OK	NO	NO	NO
Shared	OK	NO	OK	NO	NO
eXclusive	NO	NO	NO	NO	NO
Shared Intentional eXclusive	OK	NO	NO	NO	NO

Compatibility matrix for multi-granule locking

To be able to lock different granularities, we need to use a specific technique. As in the other locking mechanisms, we also need to use the structure of queues and lists per granule keeping track of waiting and locking transactions, respectively. Again the difference comes just from a new compatibility matrix with new modes. This matrix is shown in Figure 15 and its semantics are as follows:

Note

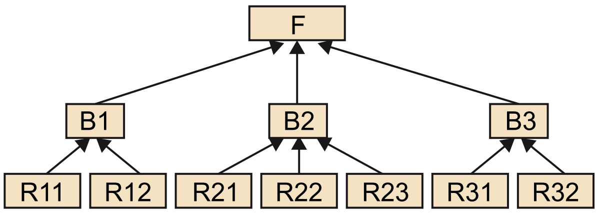
You should notice that the original compatibility matrix in the basic SX-locking is contained in this one if you only consider Shared and eXclusive rows and columns.

- Shared is used for reading a granule.
- eXclusive is used for writing a granule.
- Intentional Shared over a granule is used for reading some of its subgranules.
- Intentional eXclusive over a granule is used for writing some of its subgranules.
- Shared Intentional eXclusive is used for reading all subgranules and writing some of them.

Since an *IS* lock means we want to access some subgranules of this, it is compatible with itself, as well as exclusive access to some subgranules (i.e., *IX* locking) as soon as these sets of subgranules are disjoint, shared access (i.e., *S*) to the whole granule, or shared access to the whole granule and exclusive to some of its subgranules (i.e., *SIX*). However, it is not compatible with exclusive access to the whole granule (i.e., *X*). Regarding *IX* locking, as we just said, it is compatible with *IS*, as well as with itself, but not with any other of the locking modes. Shared access to a granule is compatible with itself and with shared access to some of its subgranules, but not with any of the others. Exclusive access, as its name indicates, is not compatible with any other kind of locking. Finally, *SIX* mode is not really necessary but just convenient (we could do the same without it). It is not compatible with itself or any other mode except *IS*.

Together with this new matrix, the concurrency control manager also has to change the protocol to automatically deal with the locks: Before accessing a granule, besides locking it, all its ancestors should also be locked in the corresponding mode.

Figure 16. Example of tree of granules



In this way, the concurrency control manager is able to deal with different kinds of granules at the same time: records, blocks, files, etc. Figure 16 shows a hierarchy with three different granularities (i.e., record, block, and file).

Figure 17. Example of schedule with multigranule locking

	T_1	T_2	T_3
1	lock(B1,S)		
2	R(R11)		
3		lock(R21,S)	
4		R(R21)	
5			lock(R22,X)
6			RU(R22)
7			W(R22)
8		lock(R23,S)	
9		R(R23)	
10	lock(B3,S)		
11	R(R32)		
12			lock(R31,X)
13		commit	🟩
14	commit		🟩
15			RU(R31)
16			W(R31)
17			commit

F			[T_1,IS] [T_2,IS] [T_3,IX]
	B1		[T_1,S]
		R11	
		R12	
	B2		[T_2,IS] [T_3,IX]
		R21	[T_2,S]
		R22	[T_3,X]
		R23	[T_2,S]
	B3		[T_1,S] [T_3,IX]
		R31	[T_3,X]
		R32	

Figure 17 shows an example of schedule with multigranule locking. Transaction T_1 locks blocks, while T_2 and T_3 lock records. Table at right hand side of the figure shows all locks for each granule after time 12. First, T_1 locks $B1$ in mode shared to be able to read $R11$ (notice that it locks the whole block to actually access only one record). The record $R11$ is not locked itself, but as a side effect, T_1 is also locking F in mode *IS* to show that it is reading some of its subgranules. Afterwards (i.e., time 3), T_2 locks $R21$ in mode shared to be able to read it at time 4. This entails the locking of $B2$ and F in mode *IS* (note that this locking is compatible with the existing one over F). Now, T_3 locks $R22$ in exclusive mode for updating it. In this case, it entails the locking of $B2$ and F in mode *IX* (again, you should note that according to the compatibility matrix in Figure 15, this is compatible with the existing locks over *B*2 and *F*). Now, T_2 locks *R*23 also in shared mode, which should be propagated upwards in the form of *IS* (which would be perfectly compatible with the *IX* lock of T_3 over *B*2), but those locks already hold. So, nothing must actually be propagated.

Afterwards, T_1 locks *B*3 in shared mode to read *R*32, which should be propagated upwards to lock *F* in the corresponding mode, but it already was locked for that transaction in the same mode. So, nothing else has to be done in this case. Finally, T_3 needs to lock *R*31 in exclusive mode to be able to update it. However, when we propagate it to the block in the form of an intentional exclusive lock, we find that *B*3 is already locked by T_1 in shared mode, which is not compatible. Thus, T_3 is queued until T_1 commits and releases the lock.

By defining a hierarchical organization of granules (and allowing to lock objects at these granularities), this concurrency control technique improves throughput, reduces structure management, and avoids some deadlock situations. The only problem is that it complicates the compatibility matrix and increments the number of structures to deal with (i.e., there are queues not only per record, but also per block, per file, etc.).

The locking granularity can be explicitly chosen by users. However, it is not a good idea to put such burden on their shoulders. Ideally, it should be the query optimizer who would do it by looking at the access plan. Depending on the percentage of a granule a transaction or statement needs to access, the system should lock the whole granule at once, or piece by piece. Nevertheless, the most common option is to dynamically scale the locking granularity. This means that the system starts locking individual records. After a given percentage of the records of a block (that can be parametrized) has been locked, the whole block is locked, and the statement locks whole blocks from here on. Then, after a given percentage of the blocks of the table has been locked, the whole table is locked, and so on and so forth. It is important to notice that lock escalation greatly increases the likelihood of deadlocks.

2.1.3. Deadlock Detection and Resolution

A deadlock with two transactions is generated when a transaction T_1 is waiting for granule G_1, which is locked by T_2, and at the same time T_2 is waiting for granule G_2, which is locked by T_1. Thus, T_1 is waiting for T_2 to end and viceversa. This can be generalized to any number of transactions showing a circular wait for dependency among them. Once this happens, the involved transactions cannot untie it by themselves, without external aid.

There are three options to handle deadlock: prevention, avoidance, and detection and resolution. The first one is unrealistic, because you should know in advance which granules the transaction will access (if it would generate a deadlock situation it would be rejected or queued before it begins). Avoidance detects hazardous situations and immediately aborts the current transaction (which might result unnecessary in the end). Therefore, in this module we will study the third possibility, where an external mechanism is needed to detect and fix the problem, which can be identified as a cycle in the Wait-For graph of the system.

There are several possibilities to look for cycles in the Wait-For graph. For example, every time a transaction is queued, we can check whether the new edge generates a cycle. This option has the advantage of having a thread to pull (i.e., the new edge), which we can assume to be part of the cycle we are looking for. Nevertheless, it can be really expensive in the presence of complex graphs.

Another alternative is to look for cycles when two transactions are queued too long, or just regularly (i.e., every x seconds). In this case, the DBMS has to analyze the whole graph. It would do it by means of the algorithm in figure 18.

Huge number of transactions

The record of the TPC-C benchmark at 2011-2-10 was approximately 30, 000, 000 transactions per minute. This means that the Wait-For graph in this case has thirty million nodes.

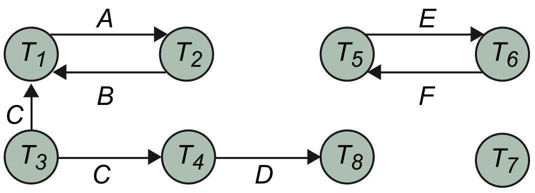
Figure 18. Deadlock detection algorithm

```
Copy the Wait-For graph
Do {
   If there is a node without successors, then remove it
} While a node has been removed
If Wait-For graph is not empty then
   Choose one node with predecessors (someone is waiting for)
   Cancel the corresponding transaction
```

First, we have to generate a copy of the graph, because we will modify it. Then, we gradually remove one by one all nodes without successors (i.e., all nodes corresponding to transactions that are not waiting for another one), together with all edges pointing to them. Eventually, either the graph will be empty (i.e., there is not any deadlock) or some cycle remains. In this later case, one of the nodes of the cycle has to be chosen and the corresponding transaction aborted.

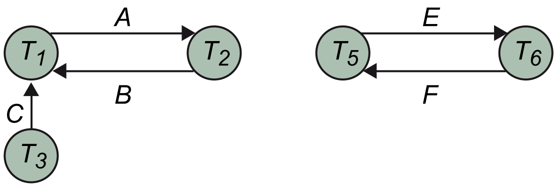
The choice will be affected by several factors: The amount of work the transaction did; the cost of undoing the changes it made; the work that remains to be done in the transaction (which would need to know what the transaction would do in the future); and the number of cycles that contain the transaction. Several policies exist to pick up one: The youngest, the shortest (up to the moment),the one that made less modifications, the one involved in more cycles, the one with less priority, etc.

Figure 19. Example of cyclic Wait-For graph



In the example of Figure 19, we have eight transactions (from T_1 to T_8) and the edges show that the tail is waiting for the head. The tag of each edge shows the granule provoking the waiting. Following the previous algorithm, in the first loop we can choose either T_7 or T_8. It does not really matter which one we choose first. Let us suppose we choose T_7. Now, in the second loop, the only node without successors is T_8, so we also remove it together with the edge pointing to it (i.e., that tagged with D). In the third loop, T_4 does not have any successor, so we remove it together with the edge tagged with C. Finally, in the third loop, there is not any node without successors. Figure 20 shows the state of the Wait-For graph at this point. Therefore, we reach step 5 of the algorithm, the graph is not empty, and in step 6 we have to choose a transaction to abort. Notice that choosing T_3 at this point would not be really smart, because it would not untie any of the deadlocks. Thus, we should choose one with an edge pointing to it (showing that someone is waiting for it). The best option would probably be T_1. After this, T_3 and T_2 can continue their execution. However, T_5 and T_6 would keep locked waiting for the next execution of the deadlock detection algorithm.

Figure 20. State of the Wait-For graph after leaving the loop of the algorithm



2.1.4. Tuning Locks

Tuning is the database administration activity of changing the parameters of the DBMS to improve its performance and throughput (i.e., transactions per minute). From the point of view of concurrency control, it is important to detect hot spots (i.e., places that everybody visits) and instruct the users to circumvent them. In this way, they avoid being locked in their queues.

If somebody has to access one of these hot spots, it should be as late as possible inside the transaction, in order to have it locked for a period of time as short as possible. This will reduce waiting time in the corresponding queue. Another idea, especially useful for insertions, is partitioning. In a one-piece table, most inserts go to its last block. By partitioning a table, we generate many "last blocks" (actually one per partition). Now we have many hot spots that will be much less crowded.

Besides this hot spot management, all DBMS provide specific mechanisms that smooth concurrency. Examples of these are read-only transactions (we will talk specifically about them in Section 3) and sequences. Although it is quite easy to implement a counter by ourselves, if we use the sequence mechanism provided by the DBMS, we avoid being locked.

Sequences

Sequences are objects defined in the standard SQL that implement a parametrized (i.e., first value, max value, increment) counter and are typically used as surrogates for the primary key of a table. Besides this flexibility, their main characteristic is that they can be concurrently accessed by many users without being locked.

Also crucial for the performance is to select the proper deadlock interval (i.e., how often the system is looking for cycles in the Wait-For graph). Small values result in a waste of time, because most times we analyze the graph to find just nothing. On the contrary, big values result in transactions in a deadlock unnecessarily waiting to be aborted and relaunched. The right value depends on the probability of having deadlocks in the system (high probability suggests small interval and viceversa).

Finally, another good practice to avoid unnecessary locking is to use DDL statements with few users logged in (or no user at all if possible). Take into account that modifying the schema of a table means locking it from head to tail. Therefore, nobody can access it at all during the execution of the corresponding statement. You should remember that creating an index, adding/removing a column or computing statistics can take quite a long time.

2.2. Timestamping

Timestamping is another concurrency control technique alternative to locking. The idea behind it is imposing an order among transactions. When a potential conflict arrives, the order between the conflicting transactions is checked. If the order is violated, the transaction during the execution of which this violation is detected is canceled.

In this case, we do not need any compatibility matrix or lists and queues associated to the granules, because nobody waits for another one to finish. All we need to keep for each granule is the timestamp of the youngest transaction reading (*RTS*) and writing (*WTS*) it. Moreover, for each transaction we keep the timestamp of the BoT (i.e., Begin of Transaction) operation. Finally, we have to modify the read and write procedures to take into account those data:

Figure 21. Timestamping read algorithm

```
procedure read(T, G) is
1: if WTS(G) ≤ TS(T) then
2:    RTS(G)= max(RTS(G), TS(T));
3:    R(G);
4: else
5:    abort(T);
6: endIf;
endProcedure;
```

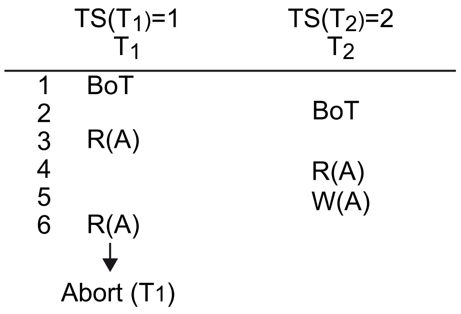
Figure 21 shows the lines of code that should be added to the read procedure. First, for a transaction *T* to read granule *G*, we have to check (line 1 of the procedure) whether the timestamp of *T* is prior or not to the timestamp of the transaction that wrote the granule. It makes no sense to let *T* read something written in its future. Therefore, if this is the case and *WTS* (*G*) is posterior to *TS* (*T*), we just abort *T* and relaunch it (you should notice that on relaunching it, we reassign its timestamp and when it tries to read *G* again, this new timestamp will not be prior to *WTS* (*G*) anymore). If everything is right, we just record (line 2 of the procedure) that the current transaction read the granule (if it is the youngest doing it), and let *T* read the granule (line 3 of the procedure). Note that if a younger transaction than the current one already read the granule we leave its timestamp in *RTS* (*G*) by taking the maximum value.

Figure 22. Timestamping write algorithm

```
procedure write(T, G) is
1: if WTS(G) ≤ TS(T) and RTS(G) ≤ TS(T) then
2:    WTS(G)= TS(T);
3:    W(G);
4: else
5:    abort(T);
6: endIf;
endProcedure;
```

Regarding the writing procedure in Figure 22, it is a bit more complex, because we have to check (line 1 of the procedure) not only whether the current transaction *T* is posterior or not to the youngest one who wrote it, but also posterior to the youngest that read it. If it is not true, as before, we cannot write the granule and we have to abort the current transaction. If the current one is posterior to the one writing and reading *G*, we just record that the current one is the youngest which wrote the granule (line 2 of the procedure) and write it (line 3 of the procedure). In this case we do not need to take the maximum value of the timestamps, because we do know that the one of *T* is greater than *WTS* (*G*), since we already checked it in the condition of the conditional statement.

Figure 23.



	$TS(T_1)=1$ T_1	$TS(T_2)=2$ T_2
1	BoT	
2		BoT
3	R(A)	
4		R(A)
5		W(A)
6	R(A) ↓ Abort (T1)	

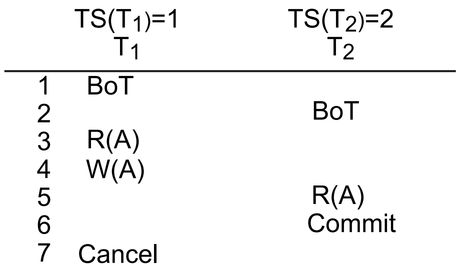
Example of timestamping concurrency control in the presence of Unrepeatable Read interference

Figure 23 shows an example of schedule with a potential unrepeatable read interference. First, T_1 reads granule A and afterwards T_2 reads and writes A. Thus, when at time 6, T_1 tries to read again the same value from A, it has been changed by T_2. Notice that it is avoided by this concurrency control technique, since T_1 would be aborted. Given that the timestamps of T_1 and T_2 equal 1 and 2, respectively, at time 3, we set $RTS(A) = 1$; at time 4 we set $RTS(A) = 2$; and at time 5 we set $WTS(A) = 2$.Therefore, when at time 6 we check whether the transaction writing A was prior or not to T_1, we find that it was not (since $WTS\ (G) > TS\ (T_1)$) and abort T_1.

Note

Notice that in the examples of timestamping concurrency control, it is important to show BoT, since it fixes the timestamp of the transaction. Opositely, it is not necessary to distinguish between R and RU anymore, because that information was only needed to choose the lock mode.

Figure 24.



	$TS(T_1)=1$ T_1	$TS(T_2)=2$ T_2
1	BoT	
2		BoT
3	R(A)	
4	W(A)	
5		R(A)
6		Commit
7	Cancel	

Example of timestamping concurrency control generating an uncommitted read

Nevertheless, as we can see in Figure 24, those modifications in read and write procedures are not enough to guarantee recoverability. As in the previous example, the timestamps of T_1 and T_2 equal 1 and 2, respectively. Now, we set $RTS\ (A) = 1$ and $WTS\ (A) = 1$ at times 3 and 4.Then, when T_2 tries to read G at time 5, there is no problem at all, because the transaction who wrote it was previous to the current one (i.e., $WTS\ (A) \leq TS\ (T_2)$).Thus, it commits at time 6, just before T_1 cancels (either rollbacks or aborts because of some reason) at time 7 generating the interference.

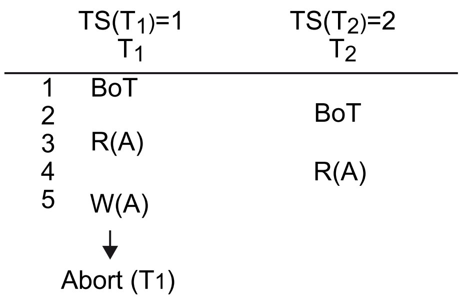
We have two alternatives to guarantee recoverability. The first alternative is to check it at commit time by making some transactions wait until others confirm. Thus, if T_2 reads a value of T_1 (being $TS\ (T_1) < TS\ (T_2)$), then T_2 has to wait for the end of T_1 (and finish in the same way, either commit or cancel). Moreover, as soon as a transaction cancels, we abort all transactions that read values written by it.

Figure 25. Modifications of timestamping algorithms to check recoverability

```
procedure read(T, G) is
if WTS(G) ≤ TS(T) and TW(G) committed then
    RTS(G)= max(RTS(G), TS(T));
    R(G);
else
    abort(T);
endIf;
endProcedure;
procedure write(T ,G) is
if WTS(G) ≤ TS(T) and RTS(G) ≤ TS(T) and TW(G) committed then
    WTS(G)= TS(T);
    W(G);
else
    abort(T);
endIf;
endProcedure;
```

Alternatively, we could check recoverability at operation time. In this case, we have to modify the previous algorithms as shown in Figure 25. We just would extend the predicate checking whether to abort the current transaction or not with a conjunction (in bold face) stating that the transaction who wrote the granule *G* already committed. So, if it did not, we abort the current transaction. As you can imagine, this second option can generate many unnecessary abortions, because it preventively aborts the current transaction even if the one that wrote *G* finally commits the change. Therefore, all along this module, we will assume that timestamping concurrency control mechanisms check recoverability at commit time and not at operation.

Figure 26.



Example of timestamping concurrency control in the presence of a potential lost update (or unrepeatable read)

Let us pay attention to Figure 26. It shows a potential lost update (or unrepeatable read) interference avoided by the concurrency control mechanism. Again, the timestamps of T_1 and T_2 equal 1 and 2, respectively. First, *RTS* (*A*) = 1, but it changes at time 4 to be *RTS* (*A*) = 2. Thus, when T_1 tries to write it at time 5, it cannot, because *RTS* (*A*)> *TS* (T_1). Therefore, T_1 is aborted and the potential interferences avoided. Nevertheless, such preventive action may be completely unnecessary, because we do not really know if T_2 was going to overwrite *A* (or read it again) or not.

Figure 27.

	$TS(T_1)=1$ T_1	$TS(T_2)=2$ T_2
1	BoT	
2		BoT
3		R(A)
4		W(A)
5		Commit
6	R(A)	

$\downarrow$

Abort (T1)

Example of timestamping concurrency control generating an unnecesary abortion

Figure 27 shows another schedule where the effect of this concurrency control technique is even worse. As in the previous examples, the timestamps of T_1 and T_2 equal 1 and 2, respectively. At time 3, T_2 sets *RTS* (*A*) = 2; and at time 4, *WTS* (*A*) = 2. Therefore, when at time 6 T_1 tries to read *A*, we find that *WTS* (*A*)> *TS* (T_1), resulting inthe abortion of the transaction. Such abortion is clearly unnecessary (because it would never generate an interference), and could have been easily avoided if the BoT of T_2 had been prior to that of T_1.

This is a pessimistic technique, but not as much as locking, because it lets more transactions continue the execution. However, the schedule must be equivalent to the serial one following the order of BoTs. From this point of view, it is not as flexible as locking and could incur in many unnecessary cancellations.

In order to avoid some unnecessary cancellations, we can delay the assignment of the timestamp to the transactions as much as possible. Concretely, we can delay it until there is a potential conflict between the current transaction and another one.

To be able to delay the assignment of timestamps, we need to keep track of the current active transactions, the read and write sets of each of them, and also inject in read and write procedures the code in Figure 28.

Figure 28. Dynamic timestamping algorithm

```
1:  foreach $T_i \in setOfActiveTx$ and $T_i \neq$ T do
2:     if ($G \in RS(T) \cap$ WS($T_i$)) or ($G \in WS(T) \cap RS(T_i)$) then
3:        if $TS(T_i)$ == $null$ and $TS(T)$ == $null$ then
4:           $TS(T_i) = maxTS + 1$; $TS(T) = maxTS + 2$;
5:           foreach $G_j \in RS(T_i)$ do $RTS(G_j) = TS(T_i)$;
6:           foreach $G_j \in RS(T)$ do $RTS(G_j) = TS(T)$;
7:           foreach $G_j \in WS(T_i)$ do $WTS(G_j) = TS(T_i)$;
8:           foreach $G_j \in WS(T)$ do $WTS(G_j) = TS(T)$;
9:        elsif $TS(T_i)$ <> $null$ and $TS(T)$ == $null$ then
10:          $TS(T) = max$TS + 1;
11:          foreach $G_j \in RS(T)$ do $RTS(G_j) = TS(T)$;
12:          foreach $G_j \in WS(T)$ do $WTS(G_j) = TS(T)$;
13:       elsif $TS(T_i)$ == $null$ and $TS(T)$ <> $null$ then
14:          assign $TS(T_i)$ so that $TS(T_i) < TS(T)$;
15:          foreach $G_j \in RS(T_i)$ do $RTS(G_j) = max(RTS(G_j), TS(T_i))$;
16:          foreach $G_j \in WS(T_i)$ do $WTS(G_j) = max(WTS(G_j), TS(T_i))$;
17:    endIf;
18: endIf;
19: endForeach;
```

Note

We are asumming that there is a variable *maxTS* containing the value of the last timestamp assigned.

According to this code (line2), when a transaction *T* reads or writes a granule *G*, we have to check if this granule belongs to the write set or the read set of another active transaction T_i. If so, we cannot delay the assignment of timestamps anymore. Thus, we have to assign them so that the current transaction is posterior to the active one that accessed *G* before (either lines 4, 10 and 14). We have three possible situations, i.e., none of them has a timestamp assigned (line 3), only the current transaction does not have a timestamp assigned (line 9), or only the timestamp of the other active transaction T_i has not been assigned (line 13). The first two cases are easier, because we can assign the next available timestamp (i.e, the maximum already assigned plus one) to the current transaction. The third case is a bit more complex, because we have to find an unused timestamp older than that of *T*. In any of the three cases, we have to propagate the new timestamps to the read sets and write sets of the corresponding transaction. As before, the third case is a bit more complex, because this may not be the youngest transaction accessing the granule (remember that timestamping techniques require that *RTS* and *WTS* contain the timestamp of the youngest transaction, respectively, reading and writing the granule), so we have to take the maximum value in order to be consistent (you should notice that, unlike the others, the assignment at line14 does not guarantee that this value is greater than those existing in *RTS* and *WTS*).

Figure 29.

	T_1	T_2
1	BoT	
2		BoT
3		R(A)
4		W(A)
5	R(A)	
6		Commit
7	Commit	

Example of dynamic timestamping

Figure 29 shows an example of dynamic assignment of the timestamps. You should compare this with Figure 27 where we assigned the timestamp at BoT. Now, at time 3, neither T_1 nor T_2 have a timestamp assigned, yet. However, both already are in *setOfActiveTx*. At this moment, T_2 reads A, so RS(T_2) = {A}. At time 4, T_2 writes A, so $WS(T_2)$ = {A}. Finally, at time 5, T_1 tries to read A, so RS(T_1)= {A} and we have to check the intersection of these sets. In doing so, we find that $WS(T_2) \cap RS(T_1)$ = {A}. Therefore, we have to assign the timestamps so that $TS(T_2) < TS(T_1)$. For example, we can do it stating $TS(T_2) = 1$ and $TS(T_1) = 2$. These timestamps have to be used to update $WTS(G)$ and $RTS(G)$ for all G in the write and read sets of T_1 and T_2. Note that, afterwards we can continue the execution of read (T_1, A) as in Figure 21 without the abortion of T_1. Finally, we remove T_1 and T_2 from *setOfActiveTx* at time 7 and 6, respectively.

Figure 30.

	T_1	T_2	T_3
1	BoT		
2	R(A)		
3		BoT	
4		R(A)	
5		W(A)	
6	R(B)		
7			BoT
8			R(B)
9			W(B)
10		R(B)	
		↓	
		Abort (T_2)	

Example of dynamic timestamping generating an unnecessary abortion

Nevertheless, dynamic assignment of timestamps does not avoid all unnecessary abortions. Figure 30 shows an example of aborting a transaction that is not avoided, while it is actually unnecessary. As in the previous example, timestamps are not assigned at BoT. It is at time 5 that we assign $TS\ (T_1) = 1$ and $TS\ (T_2) = 2$, because $A \in RS\ (T_1) \cap WS\ (T_2)$. Then, at time 9, we assign the timestamp $TS\ (T_3) = 3$, because $B \in RS\ (T_1) \cap WS\ (T_3)$ and $TS\ (T_1) = 1$. Finally, at time 10, T_2 tries to read B and finds that it was written by a younger transaction (i.e., T_3) and has to be aborted. You should notice that if we had assigned timestamps $TS\ (T_1) = 1$, $TS\ (T_2) = 3$, and $TS\ (T_3) = 2$, then the abortion had been avoided. However, in order to assign the timestamps in a smarter way, we have to keep track of the whole graph of precedences and check it for every operation, which results to be extremely expensive (more than unnecessarily aborting some transactions), or wait until the end of a transaction to check if we are able to properly assign its timestamp (which would be much more optimistic and would make abortions more expensive).

2.3. Multi-version

The idea behind this timestamping-based technique is to keep several versions of the same granule. Thus, each transaction chooses the most appropriate to read, and writings generate new versions of the corresponding granules. Obviously, as with any other concurrency control technique everything must be transparent to the end-user.

Serializability theory is adapted to be able to prove correctness of a concurrent execution of transactions with different versions of each granule. From this new point of view, as in the mono-version case, we assume that read and write operations interleave, but moreover, now we consider that read operations may access one of many available versions of a granule. The counter part of a *serial schedule* is a *standard serial multi-version schedule*, where each read operation accesses the version of a granule created by the last write operation. Following this definition, we say that a concurrent multi-version schedule is correct if it is equivalent to any standard serial multi-version schedule.

Note

You can find a brief explanation of serializability theory for multi-version concurrency control in the "Encyclopedia of Database Systems".

As in pure timestamping, we do not need any compatibility matrix, and for each transaction, we have to keep the timestamp of the BoT. Moreover, for each granule, we have to keep the different versions together with the timestamp of the youngest transaction that read it (i.e., *RTS* (G_i)), and the timestamp of the transaction that created it (i.e., *WTS* (G_i)). Again, we have to modify the read and write procedures to take into account all those data.

Figure 31. Multiversion read algorithm

```
procedure read(T, G) is
1: find $G_i$ so that
2:    $WTS(G_i) \leq TS(T) \wedge \forall j(WTS(G_j) > WTS(G_i) \Rightarrow WTS(G_j) > TS(T))$;
3: $RTS(G_i) = max(RTS(G_i), TS(T))$;
4: $R(G_i)$;
endProcedure;
```

As you can see in Figure 31, the first thing we do in lines 1 and 2 of the read operation is to find the youngest version *i* of the granule prior to the beginning of the transaction. Then, at line 3, we just record which is the youngest transaction that read G_i (i.e., either the one which did it previously or the current one), and finally, at line 4, we read it.

Figure 32. Multiversion write algorithm

```
procedure write(T, G) is
1: find $G_i$ so that
2:    $WTS(G_i) \leq TS(T) \wedge \forall j(WTS(G_j) > WTS(G_i) \Rightarrow WTS(G_j) > TS(T))$;
3: if $RTS(G_i) \leq TS(T)$ then
4:    $RTS(G_{k+1}) = TS(T)$;
5:    $WTS(G_{k+1}) = TS(T)$;
6:    $W(G_{k+1})$;
7: else
8:    abort(T);
9: endIf;
endProcedure;
```

Regarding the write operation, in Figure 32, the first two lines coincide with those in the read. Afterwards, we have to check whether a transaction posterior to the current one read G_i or not (you should notice that to update a granule, you first have to read it). If no younger transaction read the granule, it is safe to create a new version k +1 (assuming that we already have k versions) of it, initializing its read and write timestamps to that of T. Otherwise, we have to abort the current transaction.

To see how this concurrency control technique works, let us pay attention to the schedule in Figure 33, where users only work over granule A whose initial version is A_0 and initial timestamps are RTS (A_0) = 0 and WTS (A_0) = 0. We have three transactions whose respective timestamps correspond to their time of BoT. Thus, at time 2, we have to change RTS (A_0) to take value 1. Later, at time 4, it takes value 3, because T_2 reads it. Then, at time 5, T_2 creates a new version of the granule A_1, whose timestamps are RTS (A_1) = 3 and WTS (A_1) = 3. Thus, when T_1 repeats the reading of A, it actually accesses A_0, because it is the youngest one with RTS prior to 1. However, when T_3 reads the same granule, it actually accesses A_1, and changes RT (A_1) = 8, because it is the youngest transaction accessing it.

Figure 33.

	TS(T_1)=1	TS(T_2)=3	TS(T_3)=8
	T_1	T_2	T_3
1	BoT		
2	R(A)		
3		BoT	
4		R(A)	
5		W(A)	
6		Commit	
7	R(A)		
8			BoT
9			R(A)
10	Commit		
11			Commit

Example of timestamping based multiversion concurrency control committing

Figure 34 shows that, as for any timestamping-based technique, this is not enough to guarantee that there are no interferences, because it does not ensure recoverability. This example coincides with the previous one, except that T_2 now postpones its ending. When it comes at time 11, it is not a commit but a cancel (i.e., rollback or abort), which results in a read uncommitted interference for T_3. To avoid such case, we have the same possibilities as in Section 2.2 (i.e., check recoverability either at operation time or commit time).

PostgreSQL

In PostgreSQL, the database administrator has to configure how often the vacuum operation is executed. This operation is in charge of compacting the blocks of each table by removing obsolete versions.

Figure 34.

	TS(T_1)=1	TS(T_2)=3	TS(T_3)=8
	T_1	T_2	T_3
1	BoT		
2	R(A)		
3		BoT	
4		R(A)	
5		W(A)	
6	R(A)		
7			BoT
8			R(A)
9	Commit		
10			Commit
11		Cancel	

Example of timestamping-based multiversion concurrency control generating a read uncommitted interference

The clear disadvantage of multi-version is that it requires more space to store all the versions of the different granules. In most cases, the tables are periodically cleaned up to avoid unbounded growing (if this is not done and we keep all versions forever, we have a Transaction time database).

Transaction time databases

Transaction time databases do not have physical deletions or updates (these are only logical). They only insert new data, stamping them with the corresponding current time.

The advantages of multiversion concurrency control are that readings are never blocked nor rejected and writings do not have to wait for readings of the same granule.

Other side advantages appear. A technique like this facilitates recovery mechanisms (undoing changes is just a version removal). Moreover, hot backups (i.e., those that can be done without shutting down the system) are easy to implement by just copying committed versions of data previous to the starting point of the backup process.

2.4. Comparative Analysis of Concurrency Control Mechanisms

In this section, we are going to briefly analyze the different kinds of concurrency control algorithms we have seen and other possibilities that exist. We have two great families of solutions to avoid interferences:

1) Those that suspend the execution of a conflicting transaction and retake it when the threat is gone.

2) Those that ignore the threat, wait for the interference to happen and cancel all transactions involved (which includes undoing all changes done by them).

Note

Remember that a conflict appears when two transactions access the very same granule and at least one of them modifies it. Note that this does not entail an interference, but just the possibility that such problem appears in the future.

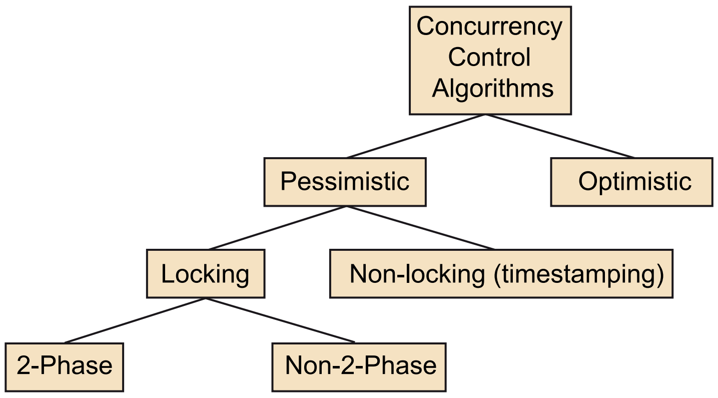
Both alternatives reduce the throughput of the system. However, the first one is pessimistic or conservative (in the sense that it suspends the execution before the interference appears, expecting that it will happen while it may not), and the second one is optimistic or aggressive (in the sense that it lets the execution continue in the presence of a conflict, expecting that it will not happen while it may). Only at some point, optimistic schedulers check whether interferences had happened, and if so, they take the appropriate measures (i.e., aborting transactions). Thus, pessimistic mechanisms synchronize transactions earlier than optimistic ones that delay the synchronization until the transaction ends. Optimistic techniques result in good performance when interferences are not common. However, when many interferences appear, they generate many cancellations, which is usually not worth.

Optimistic product

Implementations of optimistic techniques are really rare. Nevertheless, Google's Megastore (the back stage of Fusion Tables) is an example.

As depicted in Figure 35, among the pessimistic techniques, some concurrency control mechanisms are more pessimistic or optimistic than others. Locking concurrency control mechanisms are the most pessimistic techniques (in the extreme, we could exclusively lock the whole database for each transaction despite what it accesses and obviously get a serial schedule), but also the most common. Those algorithms following the 2PL protocol are characterized by the compatibility matrix they use (for example, SX, SUX or multi-granule). In this way, they range from two kinds of locks in the SX to five kinds of locks in multi-granule techniques. They can offer more or less flexibility in locking, but anyway they are the best choice in the presence of many interferences and when canceling transactions would be really expensive (even more than waiting in a locking queue). Among the locking techniques, we can distinguish those that guarantee serializability by means of the 2PL (that encompass the main stream of DBMS) and those that do not follow this locking protocol (and use instead other mechanisms like imposing an order in which transactions can lock granules to guarantee it).

Figure 35. Hierarchy of concurrency control algorithms



In between locking and optimistic techniques, we have those non-locking, appropriate for systems with not so many interferences and where canceling some transactions can be affordable. This happens when most transactions are read-only and it is rare that the granule read by one is concurrently written by another transaction. Typically, they are based on timestamping (either static, dynamic or multi-version).

From another point of view, we could distinguish two kinds of problems that may be separately addressed: synchronization among writing operations and synchronization of reads with regard to writes. In doing so, a scheduler can be designed offering different concurrency control mechanisms to deal with each one of those problems (if this is the case, it is called *hybrid*).

A usual compromise in commercial systems is to use multi-version timestamping for read-only transactions, and Strict-2PL for read-write transactions. In this way, read-only transactions are not delayed by locks and, at the same time, never have to be aborted, because they always read the appropriate version to their timestamp.

2.5. Transaction Chopping

From the point of view of concurrency control, long transactions degrade performance, no matter the concurrency control technique you use. On the one hand, if you are using locks, other transactions wait too much, because the long one will retain locks too much (due to Strict-2PL). Moreover, a long transaction (which probably accesses many granules) will probably have to wait, because the chances of finding one of those many granules locked is high. Since results are not visible until it commits, even if the transaction was queued at the very last granule it accessed, it will look like the first change also took a lot of time to be done. On the other hand, if you are using timestamps, the probability of being aborted increases with the length of the transaction, because the likelihood of accessing a granule already accessed by a younger transaction is higher as time goes on.

In some cases, we can split the transaction into different pieces. However, you should note that chopping a transaction implies releasing the locks in the break point or reassigning the timestamp. Before doing it, we should ask ourselves whether this transaction can interfere with others, whether others can interfere with this and, in the end, whether it really matters. Typically, user interaction is performed outside the transaction, even if it may generate some interferences.

Being able to chop a transaction and how to do it depends not purely on the transaction. It depends on the other transactions in the system being executed concurrently and the isolation level we need to guarantee.

Note

Tickets selling applications actually use two transactions along the process: The first one shows the empty seats and, after choosing one, the second transaction really books the choice and manages the payment. Thus, other clients may interfere the execution (e.g., choosing the same seats you see that are empty before you do it).

Transaction chopping is a technique to improve the throughput of the system generating bottlenecks by splitting transactions into different pieces without generating interferences. You should notice that we can freely chop transactions if the following conditions are true:

1) All existing transactions (maybe parametrized) are known in advance.

2) We only want to guarantee `REPEATABLE READ` (not `SERIALIZABLE`).

3) We know when a transaction may execute a `ROLLBACK`. It is even better if we not only identify were rollbacks can appear, but can also move them towards the BoT.

4) In case of system failure, we could know which pieces of a transaction committed and which did not, in order to redo the remaining pieces (and the application can take care of this).

Fist of all, we need to define some concepts. A chopped transaction is *rollback-safe* if it never rolls back or if this can only be done in the first piece. We say that a design is *rollback-safe* if all its transactions are rollback-safe. Besides these concepts, we also need to draw a *Conflict-Sibling* graph. In this graph, we have one vertex per transaction piece; one edge of type *S* between pieces of the same transaction; and one edge of type *C* between pieces of different transactions that may generate an interference (i.e., they may act over the same data, and at least one of them writes).

A transaction design is correct if it is rollback-safe and its Conflict-Sibling graph (CS-graph) does not have a mixed cycle (i.e., a cycle involving some *S* and some *C* edges).

Figure 36. Possible topologies for a CS-graph of size 3

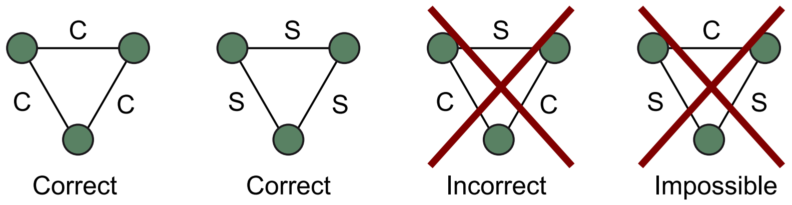
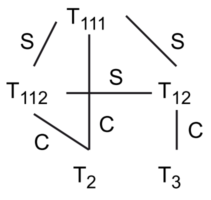


Figure 36 shows all possible configurations of CS-graphs of three vertices. Let us analyze them from left to right. The first one corresponds to three pieces of three different transactions. It does not generate any problem, because the usual concurrency control mechanisms will take care of the potential interferences. The second one corresponds to tree pieces of the same transaction. This does not generate any problem, because all pieces belong to the same transaction and therefore acting over the same granule does not result in an interference. However, the third one is a problem, because the cycle shows a mix of edges. The bottom vertex conflicts with two different pieces (i.e., siblings T_{11} and T_{12}) of the same transaction (i.e., T_1). If we assume, for example, a locking mechanism for concurrency control, the locks would be released at the end of T_{11} introducing a potential interference, while keeping them together would avoid any possible interference, because the granules would remain locked from the beginning of T_{11} until the end of T_{12}. Therefore, the behavior of the transaction would be potentially different if it were chopped or if it were not. Finally, the right-most graph depicts an impossible configuration, because if piece *A* is a sibling of *B* and at the same time *B* is a sibling of *C*, then *A* and *C* are also siblings and cannot show any conflict between them.

Figure 37. Example of CS-graph with interference

```
     T111    T112       T12
T1: R(x) | W(x) | R(y) W(y)
T2: R(x)  W(x)
T3: R(y)  W(y)
```

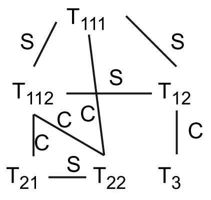


We can consider the three transactions in Figure 37 (i.e., T_1, T_2, and T_3). They would never generate an interference in the presence of a scheduler appropriately configured. However, let us analyze what happens if we chop T_1 into two pieces (i.e., T_{11} and T_{12}) and then T_{11} is chopped again into two pieces (i.e., T_{111}, T_{112}) for a total of three pieces (i.e., T_{111}, T_{112}, and T_{12}). Considering that we are using a locking concurrency control mechanism, at the end of each one of these pieces, locks are released. Thus, for example, T_2 could be executed between T_{111} and T_{112}, resulting in a lost update. This risk would be pointed out by the CS-graph in the right, where we can see that there is a mix cycle involving the two pieces of T_1 (i.e., T_{111}, T_{112}) and T_2.

Figure 38. Example of CS-graph chopping two transactions at the same time

```
    T111   T112       T12
T1: R(x) | W(x) | R(y) W(y)

    T21     T22
T2: R(x) |  W(x)
T3: R(y)    W(y)
```

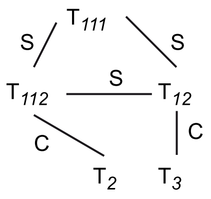


An interesting property we can analyze is that once there is a mixed cycle, it will never disappear by generating more pieces of any of the involved transactions. For example, as depicted in Figure 38, if we chop T_2 into two pieces (i.e., T_{21} and T_{22}), a new sibling edge would appear between them. Now, the original cycle contains four vertices instead of just three, and has two sibling edges instead of only one (the two conflicting edges remain between T_{21} –T_{112} and T_{22} –T_{111}). Moreover, a new cycle appears between T_{21} –T_{112} –T_{22}.

Another important property is that chopping a transaction does not affect chopping others. As we explained above, once a cycle appears, chopping a transaction does not solve the problem. The opposite is also true. If two transactions (i.e., T_1 and T_2) chopped into pieces and a mix cycle exists, collapsing all pieces of one of them (i.e., T_2) into a unique transaction does not make the cycle disappear. Therefore, without loss of generality and for the sake of simplicity, we can chop each transaction one by one considering all the others as a whole.

Figure 39. Example of CS-graph without interferences

```
    T111   T112       T12
T1: R(x) | W(x) | R(y) W(y)
T2: R(x)
T3: R(y)    W(y)
```



The example in Figure 39 shows the same transactions as the previous one, except that T_2 is read-only. If we pay attention to the corresponding CS-graph, we can see that the mix cycle has disappeared. The only existing cycle now is homogeneous (all are sibling edges). This means that no potential interference has been introduced by chopping T_1 into those three pieces. You should notice that T_1 has not changed from one example to the other. Consequently, we can say that whether to chop a transaction or not does not depend on itself, rather on the others.

Figure 40 shows the procedure we follow to chop a transaction into pieces. Actually, it is not necessary to explicit sibling edges. We can just pretend that all pieces of a transaction are connected by them (depicting them will only complicate the graph by adding a clique).

Clique

A graph (or subgraph) where each vertex is connected to all others is called a clique.

Figure 40. Transaction chopping algorithm

1) Generate a piece and place there all writings that may be rolled back.

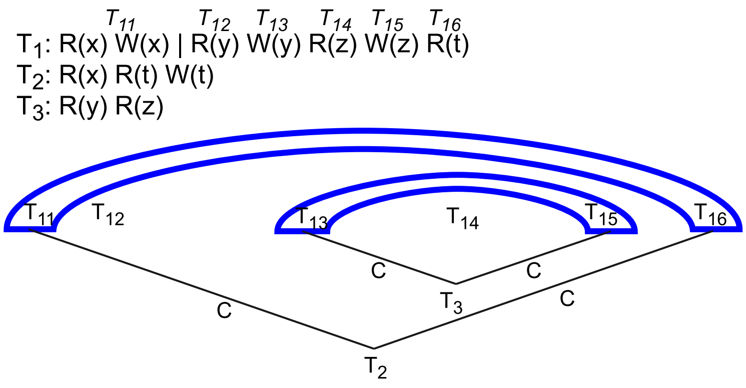
2) Generate a piece for each data access not in the first piece.

3) Generate the CS-graph (drawing only C-edges) considering those pieces and all other transactions as a whole.

4) Mark all pieces in the same connected component of that graph.

5) Check the precedences between the resulting pieces and join them as necessary.

Figure 41. Example of transaction chopping reordering the operations



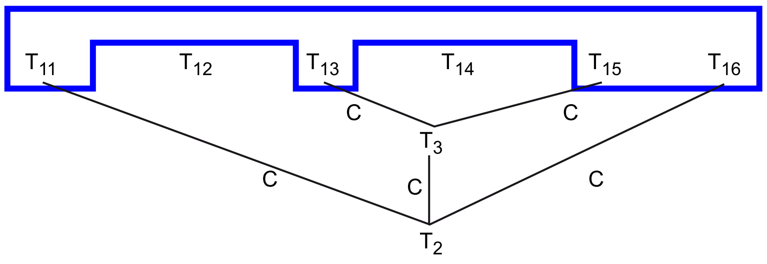
Let us explain how the procedure works by looking at the example in Figure 41. We want to see if it is possible to chop T_1 into pieces. We could do the same for T_2 and T_3, but as we said, it is not necessary to chop the three of them at the same time, because the result absolutely coincides and it only complicates the process. The first thing we have to do is finding the first piece of T_1 (i.e., T_{11}). In order to do that, we should check where are the cancel buttons in the application. Let us assume that in this case, the user has the possibility of rolling back T_1 only after the first writing. Therefore, all previous operations belong to the first piece (step 1). In the second step, for any other operations in the transaction, we suppose that they conform a piece each (step 2). Afterwards, in the third step, we have to draw all C edges. In this case, we find four: $T_{11} - T_2$, $T_{13} - T_3$, $T_{15} - T_3$, and $T_{16} - T_2$. In step four, we see that these edges connect T_{11} with T_{16}, and T_{13} with T_{15}, which means that both pairs should be in the same pieces. Thus, T_{16} should go to the first piece together with T_{11}, and T_{13} and T_{15} would be the fourth piece at the end of the transaction (you should notice that pieces T_{12} and T_{14} cannot be left at the end of the transaction, because they read the granules that are later written by T_{13} and T_{15}, respectively). Therefore, after step five, it results in T_1 being chopped into four pieces: $R(x)\ W(x)\ R(t)$, $R(y)$, $R(z)$, and $W(y)\ W(z)$.

In Figure 42, as in the previous example, in the first step, we find that to be rollback-safe, the first piece must contain the first write operation. Then, in the second step, we would generate the other five pieces from T_{12} to T_{16}. In the third step, with regard to the previous example, we have added two operations to T_2 that generate a new C-edge in the graph (you should notice that a conflict pops up between T_2 and T_3).

Figure 42. Example of transaction chopping keeping the ordering of the operations

T_{11} T_{12} T_{13} T_{14} T_{15} T_{16}

T_1: R(x) W(x) | R(y) W(y) R(z) W(z) R(t)
T_2: R(x) R(t) W(t) R(v) W(v)
T_3: R(y) R(z) R(v) W(v)



Thus, in the fourth step, we would find out that adding the new edge to those already found in the previous example, T_{11}, T_{13}, T_{15}, and T_{16} result to be connected. Finally, in the fifth step, we see that now they have to belong to the same piece, but T_{11} must be in the first piece to remain rollback-safe, and T_{12} and T_{14}, as before, cannot be left at the end of the transaction (because we are assuming that writes are always preceded by the corresponding reads). Therefore, the conclusion is that adding new operations to T_3 results in not being able to chop T_1.

3. Transactions in Oracle

In this section we are going to explain how Oracle 10g implements concurrency control, which is actually a hybrid system. Take into account that no commercial DBMS provides the details of its implementation. Thus, what we explain here comes from what the manuals say and what we can see trying the corresponding SQL sentences.

The first thing we have to say is that Oracle does not guarantee serializability, but only snapshot isolation (this prevents phantoms but still allows some rare cases of inconsistent analysis).

Snapshot isolation

This is a weaker version of serializability implemented in some DBMS, for example Oracle.

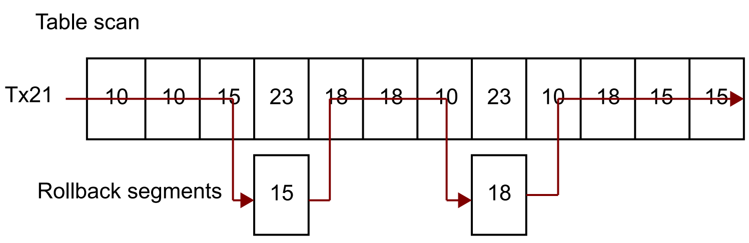
Basically, it offers two parameters to configure the behavior of transactions from the point of view of concurrency, namely read consistency and isolation level. Each of them has two different values. Let us analyze each one of them separately.

Oracle DBMS defines two kinds of read consistency: statement-level and transactions level. In any case, read operations do not generate any kind of lock over data. This means that if we use statement-level read consistency (which is the default), the DBMS only guarantees that all data accessed by a query correspond to the same snapshot of the database. However, if later on in the same transaction, we execute the same query again, it may return a different result (i.e., we may have unrepeatable read or phantom interferences). On the other hand, with transaction-level read consistency, all read operation in the transaction will see data corresponding to the same snapshot of the database (i.e., unrepeatable read and phantom interferences will be avoided). Transaction-level read consistency is guaranteed for read-only transactions (defined by means of `SET TRANSACTION READ ONLY;`), or if you choose serializable isolation level (see below).

In both cases, in order to obtain this effect without locking, either statement or transaction read consistency modes the DBMS uses multi-version concurrency control and the timestamp is assigned at the beginning of the statement or the transaction, and writing transactions use strict 2PL, using private copies similarly to those explained in Section 2.1.1. The *TS* of the transaction is assigned to the *WTS* of the granule's version at commit time, when the private copies are published.

Those copies are dynamically managed thanks to the information stored in the rollback segments. The rollback segments are created by means of the statement `CREATE ROLLBACK SEGMENT <name>;` and contain the old values of data that have been changed by uncommitted or recently committed transactions. A transaction can be explicitly assigned to a given rollback segment by means of the statement `SET TRANSACTION USE ROLLBACK SEGMENT <name>;`.

Figure 43.



Usage of rollback segments in Oracle for multiversion concurrency control. They are objects that Oracle 10g uses to temporally store the data necessary to undo changes made by transactions.

Figure 43 sketches how a transaction *Tx* with timestamp 21 that tries to scan a table would use the rollback segments where other transaction were assigned to provide the right snapshot of that table. Each rectangle would correspond to a block of the table and the number inside would be the *WTS* of the block. Thus, the first three blocks would be retrieved from the table itself. However, the *WTS* of the fourth is posterior to *TS* (*Tx*). Therefore, we would retrieve that block from the rollback segment where the transaction that wrote it was assigned. The same would happen for the eighth block.

From the point of view of the isolation level, Oracle only uses two out of the four in the standard, namely `READ COMMITED` (by default) and `SERIALIZABLE`, which can be chosen by means of the statement `SET TRANSACTION ISOLATION LEVEL [READ COMMITTED|SERIALIZABLE];`. Nevertheless, the name of the level should not mislead you about its consequences. If you chose serializable, it is still possible for transactions to interleave in ways that make the data invalid according to some business rule which is obeyed by every transaction running alone (i.e., inconsistent analysis interferences may appear).

READ COMMITTED

`READ COMMITTED` being the default and minimum isolation level is a consequence of using multi-version concurrency control, because it guarantees that only commited versions are accessed.

> Let us suppose that a bank gives a bonus of 0.1% of the balance of each account if it is greater than 1,000 €, but this is always added to another account. For example, if $A >$ 1,000 then $B := B$ +0.1A. The same works for B, and if $B >$ 1, 000 then $A := A$ + 0.1B. If these two transactions are executed concurrently with snapshot isolation the result will be wrong. Imagine A = 1,000,000 and B = 1,000,000. Now, T_1 reads A and T_2 reads B. Since both are greater than 1,000, we should add 1,000 to both A and B. However, if we execute first T_1, it will add 1,000 to B. Executing T_2 afterwards will read 1,001,000 from B and will add 1,001 to A, which does not correspond to the result of the concurrent execution.

Both `READ COMMITED` and `SERIALIZABLE` levels provide exclusive locks for record modification. Therefore, both will wait if they try to change a row updated by an uncommitted concurrent transaction. However, Oracle only allows a serializable transaction T_1 to modify a record if this was modified by a transaction T_0 that committed before T_1 began. In other words, it generates an error when T_1 tries to modify a record that was changed (updated or deleted) by a transaction that committed after the BoT of T_1. If the isolation level is

`READ COMMITED`, T_1 is allowed to continue its execution after T_0 commits, while in `SERIALIZABLE` mode, T_1 only continues its execution if T_0 is canceled. This is done this way to avoid lost update interferences.

Although DML only locks records, Oracle 10g distinguishes two different granules: table and record. However, it never scales locks (you can lock all records in a table one by one, and the granularity will not automatically change to lock the table as a whole). Table locking is mainly used for DDL. Most DDL statements lock tables in exclusive mode, but some only need shared. Moreover, `SELECT ... FOR UPDATE` statements also lock tables but in intentional shared mode, until some tuples are modified, that changes the lock mode of the table to intentional exclusive. This default behavior for table locking can be modified by the user by means of the statement `LOCK TABLE <name> IN [SHARE|EXCLUSIVE] MODE;`.

Read-for-update in SQL

This statement in the definition of a cursor tells the DBMS that we want to read a granule to modify it later on.

Since Oracle does not lock data for reading, neither scales locking (except when explicitly stated by the user), deadlocks occur infrequently. However, infrequent does not mean impossible and Oracle implements a detection and resolution policy. Every time a granule is locked, the Wait-For graph is checked, and if a cycle exists, an error message is returned to the user. The user can then relaunch the same statement again after waiting or just rollback the whole transaction.

4. Distributed Concurrency Control

In this section, we are going to explain how to deal with concurrency in a distributed environment. As before, we assume that the distributed DBMS (DDB-MS) is fully reliable (not experiencing hardware or software failures), and data are not replicated. Nevertheless, we will see that the problem becomes much more complex than in a centralized DBMS. First, we should clearly state what a distributed database (DDB) is and where difficulties come from:

> "A distributed database is a collection of multiple, logically interrelated databases distributed over a computer network."
>
> Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.*

Figure 44. Concurrency control in distributed databases

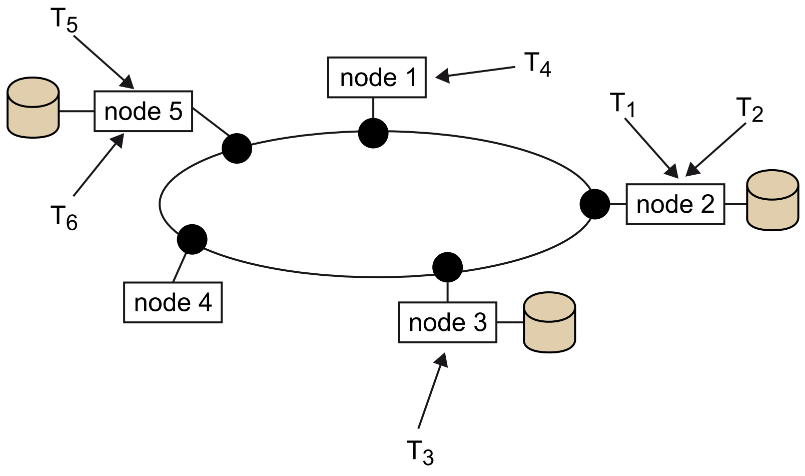


Figure 44 sketches the consequences of that definition. On the one hand, we have a network where data is spread over different nodes (i.e., 2, 3, and 5, in this case). Moreover, users can issue transactions also from different nodes in the network (independently of whether they contain data or not). Therefore, transactions posed by users could be solved locally, or need some data in other nodes.

Typically, distributed databases are classified depending on their heterogeneity and autonomy. Heterogeneity comes in the form of system as well as semantics heterogeneity, while autonomy ranges from one DBMS managing distributed data to several loosely coupled DBMSs. In this module, we will only deal with the easiest case: homogeneous and tightly coupled DBMSs without any kind of autonomy. Just imagine that we want to deal with some kind of autonomy. Then, we may have two different kinds of users, either global and local users, and confidentiality issues would arise, among others.

The most important characteristic of a truly DDBMS is transparency. It refers to hiding from the user the problems of implementing data and processing distribution. This clearly affects to data retrieval, but also ACID properties generate some difficulties that must be hidden.

ACID transactions

ACID transactions guarantee the following properties: Atomicity, Consistency, Isolation and Durability.

We have to guarantee global atomicity, recoverability and serializability. Let us pay attention now to the latter.

Figure 45. Global interference

Node$_1$ (DB$_1$)		Node$_2$ (DB$_2$)	
T_1	T_2	T_1	T_2
R(X)			
W(X)			
	R(X)		
			R(Y)
		R(Y)	
		W(Y)	
		C	
C			
	C		
			C

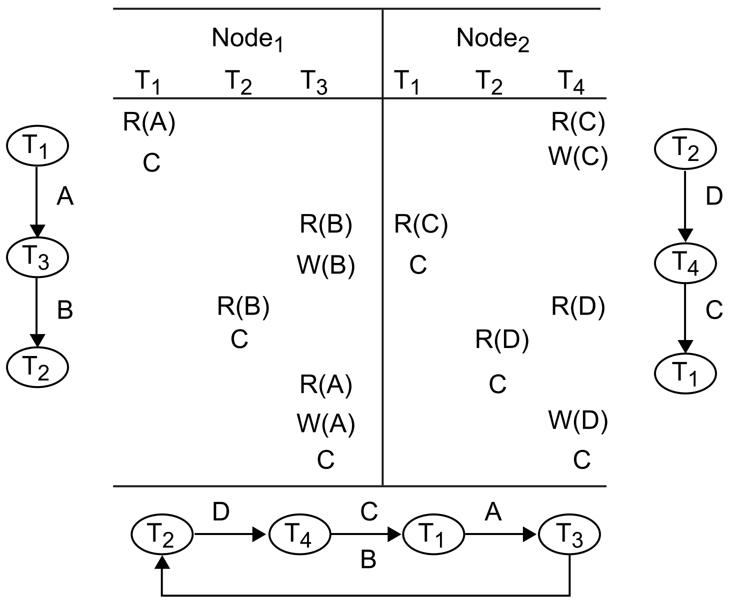
Imagine we have a distributed system composed of the machines of several banks, where each bank hosts the data of its corresponding accounts. Let us suppose now that we have two transactions being executed concurrently. They are depicted in Figure 45. The first transaction T_1 transfers 100 € from account X at site $Node_1$ to account Y at site $Node_2$. The other transaction T_2 reads both balances and adds them. It may happen that T_1 substracts 100 € from X; then T_2 reads X and Y (from the corresponding site); and finally T_1 adds 100 € to Y. If we only look at DB$_1$, there is not problem, because a transaction wrote X and afterwards another one read X. If we only look at DB$_2$, there is no problem either, because a transaction read Y and afterwards another one overwrote it. Thus, the local concurrency control mechanisms will not avoid any interference, because their partial view does not show it. Nevertheless, the result of T_2 in that case would show 100 € less than it should, because the quantity was substracted but not yet added by T_1 when T_2 was executed.

Figure 46, shows a more complex example where local concurrency control mechanisms would not detect the global interference. Graphs at both sides of the schedule show local precedences between transactions (left hand side for $Node_1$ and right hand side for $Node_2$). At $Node_1$, T_1 has to precede T_3, because T_1 must read A before T_3 writes it; and T_3 has to be executed before T_2, because T_3 has to write B before T_2 reads it. At $Node_2$, T_2 has to precede T_4, because T_2 has to read D before T_4 writes it; and T_4 has to be executed before T_1, because T_4 has to write C before T_1 reads it. Since both graphs are acyclic (actually linear), we may wrongly conclude that there are no interferences. However, they only reflect a partial view. If we pay attention to the global precedence graph under the schedule, we can see that it contains a cycle.

Precedences graph of a schedule

This is a graph where nodes correspond to transactions and there is an edge between two transactions from T_1 to T_2 if T_1 must appear first in any serial schedule equivalent to this schedule.

Figure 46. Global serializability



Therefore, we do have global interferences among T_1, T_2, T_3, and T_4. The complexity of this case comes from the fact that besides global transactions T_1 and T_2, we also have local transactions T_3 and T_4 that may not be known by the global system.

We should conclude that a global concurrency control mechanism is necessary to guarantee global serializability. Local ones are not enough if they are not working coordinately.

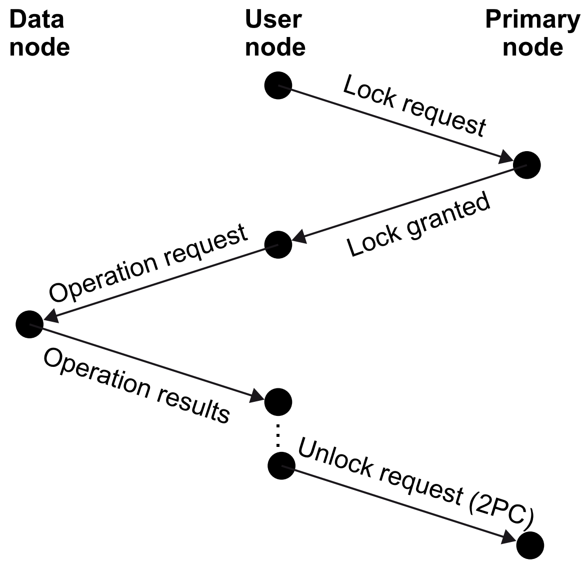
Note

Remember that serializability theory stablishes that if the precedences graph is acyclic, we can guarantee that there is no interference among the transactions.

4.1. Distributed Locking

The Strict-2PL protocol we saw in Section 1.3. can be easily extended to DDBMSs. The way to do it is centralizing the concurrency control in one single site.

Figure 47. Centralized Strict Two-Phase Locking (2PL)



Two-phase commit

"2PC protocol is a distributed algorithm to ensure the consistent termination of a transaction in a distributed environment."

Jens Lechtenborger. *Encyclopedia of Database Systems*

Messages

Arrows depict messages sent from one node in the distributed system to another.

Figure 47 sketches this possibility. There is one primary node which knows all granules in the DDB. It is the only one locking and unlocking, and keeps track of all locks. Before performing any operation over data, the user node has to be granted by the primary node. Moreover, when the user node finishes the transaction, it has to contact again the primary node to release the locks.

The main advantage of such protocol is that it is easy to implement and deadlocks are easy to detect (as easy as in a centralized DBMS, since the concurrency control is actually centralized despite data being distributed). However, the disadvantages are that if the primary node is down, so is all the DDBMS, which clearly compromises the reliability and availability of the system as a whole. Moreover, the primary node may be overloaded to the level of becoming a bottleneck in the presence of many transactions, the algorithm has a high communication cost, and implies a loss of autonomy for the participants.

An alternative to centralize the concurrency control is to also distribute it as we do with data. Thus, we will have lock managers at each site implementing distributed 2PL. For example, each node could manage the locks of those granules physically stored there.

Figure 48. Distributed Strict Two-Phase Locking (2PL)

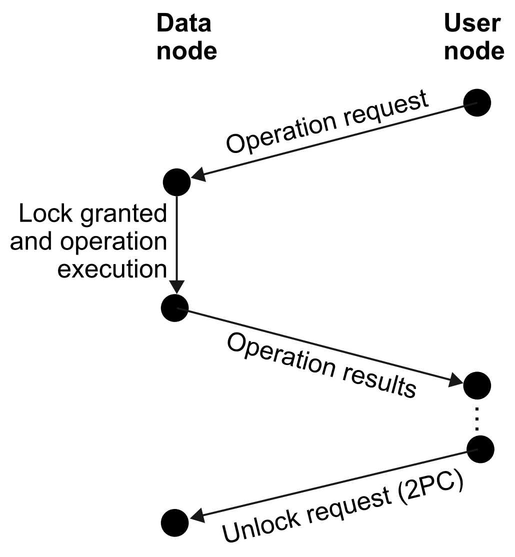


Figure 48 sketches this possibility, in which there is no primary node (i.e., every node can grant access to some granules). Now, when the user node accesses the data, the corresponding node locks them. It is the responsibility of the user node to communicate the end of the transaction to all nodes whose data were accessed throughout it.

Note

You should notice that with a distributed concurrency control mechanism if any node cannot be reached, the other can keep on working normally, while in the centralized one, if the primary node cannot be reached the whole system is down.

The main advantages of distributed locking are that concurrency control overload is distributed, resulting in a reduction in the communication costs and being more resistant to failure. On the contrary, it is harder to implement (specially deadlock detection), and we need some mechanism (e.g., timeout) to ensure that in case of user node or communications failure in the middle of a transaction, data do not remain locked forever.

4.1.1. Deadlock Detection

As in any locking concurrency control mechanism, we have to detect deadlocks. Again, doing it locally is not enough and we need to do it globally. A global Wait-For graph is necessary.

Figure 49. Example of distributed deadlock

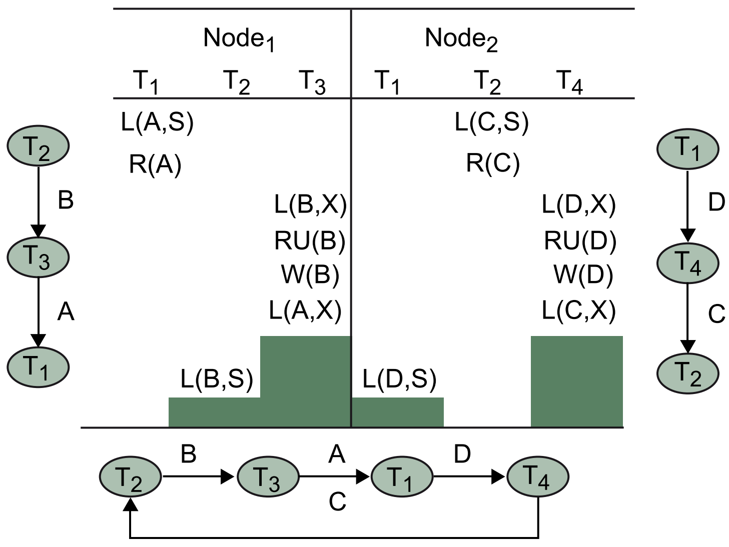


Figure 49 shows an example where local Wait-For graphs are not enough. At *Node*$_1$, T_2 is waiting for T_3 to release the lock over *B*; and at the same time T_3 is waiting for T_1 to release the lock over *A* (this is reflected by the left hand side graph). At the same time, at *Node*$_2$, T_1 is waiting for T_4 to release the lock over *D*; and this is waiting for T_2 to release the lock over *C* (this is reflected by the right hand side graph). Both graphs are linear and do not show any cycle. Therefore, no local deadlock is detected. Nevertheless, as you can see in the global Wait-For graph depicted under the schedule, T_2 is waiting for T_3 to commit, T_3 is waiting for T_1, T_1 is waiting for T_4, which in turn is waiting for T_2 closing the loop and generating a typical deadlock situation that should be detected and broken. As in the previous example, we have two global transactions (i.e., T_1 and T_2) and two local ones (i.e., T_3 and T_4), which makes the problem more difficult to solve.

As before, we could implement centralized deadlock detection even if the system is distributed, by just choosing a primary node. All other nodes should send to this their Wait-For graph (actually only changes in the local graph shave to be sent to the primary node), which would be integrated there. The advantage of such implementation, as before, is simplicity, but the well-known disadvantages appear (i.e., potential bottleneck, availability and reliability).

Alternatively, we have two options to implement distributed deadlock detection: Send the whole graph to all nodes (which is really expensive and the same deadlock would be detected many times generating potentially contradictory decisions from different nodes), or just send the right piece of graph to the right node (which is cheaper and only one node would detect a deadlock, avoiding any kind of coordination for decision making).

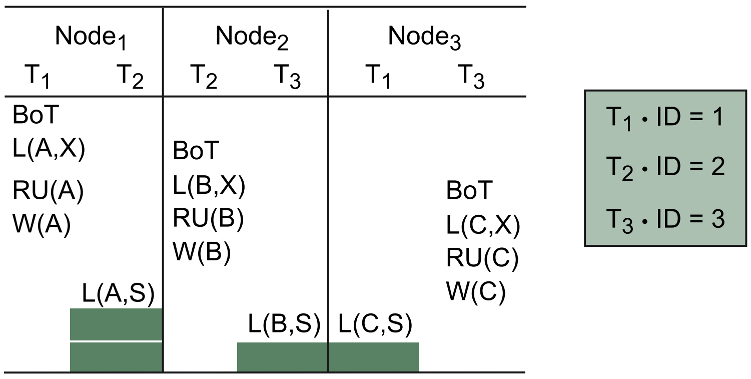
In order to implement distributed deadlock detection we need globally unique transaction identifiers (for transaction T_i, we will note it $T_i.ID$). Then, each node keeps its local Wait-For graph, where vertices are enriched to represent either transactions or nodes. A vertex representing a node in the DDBMS (and the corresponding edge) is added to the local graph when a distributed transaction is waiting for results from another node (the node coordinating the user transaction should provide this information).

To detect a deadlock, each node checks its Wait-For graph either regularly (at a given interval) or when gets a piece of graph from another node. On checking its Wait-For graph, the node should send to each node N_x every path $T_i \rightarrow \cdots \rightarrow T_j \rightarrow N_x$ so that $T_i.ID < T_j.ID$ and transaction T_j is active in the node. On receiving a piece of graph from another node, this is collapsed with a copy of the local one. As in the centralized case, if there is a cycle in the graph, there is a deadlock and the corresponding actions should be taken.

Distributed deadlocks in Oracle 10g

In distributed transactions, local deadlocks are detected immediately by analyzing the Wait-For graph every time we add an edge, and global deadlocks are detected just by a simple time out, without analyzing the Wait-For graph. Once detected, non distributed and distributed deadlocks are handle in the same way.

Figure 50. Example of distributed deadlock detection schedule



Node$_1$		Node$_2$		Node$_3$	
T_1	T_2	T_2	T_3	T_1	T_3
BoT					
L(A,X)		BoT			
RU(A)		L(B,X)			BoT
W(A)		RU(B)			L(C,X)
		W(B)			RU(C)
	L(A,S)				W(C)
			L(B,S)	L(C,S)	

$T_1 \cdot ID = 1$
$T_2 \cdot ID = 2$
$T_3 \cdot ID = 3$

To better explain how this works, let us go through the schedule in Figure 50. In this example, we only have three transactions (i.e. T_1, T_2 and T_3, whose identifiers are 1, 2 and 3, respectively) being executed concurrently in three nodes. All them lock exclusively a granule (*A*, *B* and *C*, respectively) and try to get a shared lock on one previously locked by another one (*C*, *A* and *B*, respectively). Thus, all end being locked at the same time.

Figure 51. Example of distributed Wait-For graph

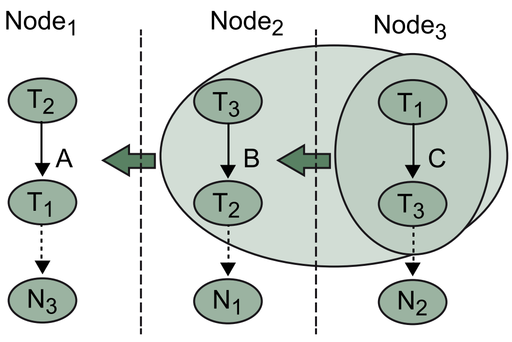


Figure 51 shows the Wait-For graphs at each site and how they are sent from one to another. First, *node*$_3$ would detect a problem, because T_3 is waiting for another node and the identifier of T_1 is less than that of T_3 (you should notice that it is the only one where this happens, because in the other two, the identifiers of the transactions in the path do not fulfil that condition). Therefore, the whole path is sent from *node*$_3$ to *node*$_2$ that would merge it with its own Wait-For graph, resulting in that T_1 is waiting for T_3 that in turn is waiting for T_2. Thus, again the identifier of the transaction at the tail of the path (i.e., T_1) is less than that of the transaction at the head (i.e., T_2), and this whole path is sent to *node*$_1$.

Figure 52. Example of reconstructed Wait-For graph at Node1

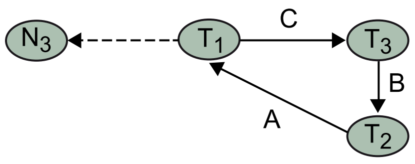


Figure 52 shows the final Wait-For graph obtained at *node*$_1$ after the local one is merged with the one obtained from *node*$_2$. Now, it is clear that it contains a cycle. Consequently, *node*$_1$ will take some action to break the deadlock.

See also

The same actions can be taken to break a distributed deadlock as those explained in Section 2.1.3. for the breaking of local deadlocks.

4.2. Distributed Timestamping

Centralized locking and deadlock detection seriously compromise availability and reliability. As discussed above, distributing these algorithms solves those problems but complicates the implementation and management and, to some extent, still results in a loss of autonomy of the different components of the DDBMS.

As in centralized systems, using locking is not necessary to guarantee serializability. We can also use timestamping mechanisms. Luckily, the distributed timestamping concurrency control, works exactly as the local version.

The only difference is that globally unique and monotonically ordered timestamps are needed. This means that all timestamps must have a different value and if one transaction began after another one, the timestamp of the former will be greater than that of the latter. One way to implement it would be using a global counter maintained in a given site. However, all sites should communicate to this before beginning a transaction, negatively affecting the performance, reliability and availability of the whole system.

Identifiers vs timestamps

The difference between having a transaction identifier and a timestamp is that, while both uniquely identify the transactions, only the timestamp shows an order between them (i.e., it is monotonically ordered).

Alternatively, we should implement some kind of distributed mechanism to synchronize the assignment of timestamps. This implementation is based on the interchange of messages between nodes and the following axiom:

"A message always arrives after it was sent."

Lamport clocks axiom

The distributed generation of timestamps is based on Lamport clock axiom, which is reflected in that each timestamp assigned to a transaction T_i (i.e.,$TS(T_i)$) must contain two attributes: local time $TSN(T_i).T$ and originating node $TS(T_i).N$. Moreover, we synchronize the clocks of two machines with every message exchange. Thus, the clock must be managed as follows:

Messages

They are the mechanism used for communication between different nodes, Figures 47 and 48 show some examples.

- Every local action (i.e., read, write, commit, etc.) increases the clock (i.e., two actions cannot be done in less than one time unit).
- Every external message sets the local clock C to max (C, $TS(T_i).T$+1), where T is the transaction generating the message.

Let us suppose that we have two nodes (i.e., $Node_1$ and $Node_2$), at $Node_1$, it is time 5 and at $Node_2$ it is time 2. If we execute two operations at $Node_1$, the first one will be executed at time 5 and the second one at time 6. If the first one takes less than one time unit, we will increment the clock to force the second one to be executed at time 6. Now, we send a message from $Node_1$ to $Node_2$. This message contains the TS of the corresponding transaction at $Node_1$ (i.e.,$TS(T_i).T$). If when this message reaches $Node_2$, the time at this node is less than that in the message, we reset the clock at $Node_2$ to take the value $TS(T_i).T$ +1.

The only difference between centralized timestamping concurrency control and the distributed version is that in the distributed version, two timestamps must be compared as follows:

$$TS(T_1) < TS(T_2) \Leftrightarrow TS(T_1).T < TS(T_2).T \vee (TS(T_1).T = TS(T_2).T \wedge TS(T_1).N < TS(T_2).N)$$

The following table contains examples of timestamp comparison:

TS(T_1)		**TS(T_2)**		**Comparison**
T	N	T	N	TS(T_1) < TS(T_2)
4	1	5	2	true
5	1	4	2	false
4	1	4	2	true
4	2	4	1	false
4	1	4	1	impossible

5. Replica Management

In this section, we get rid of the simplification we made when considering that data were not replicated. In a DDBMS, we would like users' queries to be answered with only those data in the user node. Transferring data from one site to another is really expensive, and replication of data to benefit from locality is a solution to this. Another advantage of replication is availability of the system and resilience to failures (one copy might be enough to keep on working).

Locality of data

A well-known optimization technique in a distributed system is to process locally as much data as possible, before shipping it through the network.

Nevertheless, the price we have to pay for those advantages is that we have to maintain the consistency of the different copies of data distributed along the system. With a unique copy, all changes are done on it. However, in the presence of replication, we can adopt different policies to guarantee that all copies contain exactly the same data (i.e., they are consistent). Thus, we are going to study how to deal with the consistency of replicas in a distributed environment, considering that it must be transparent to the users.

Figure 53. The four cases of replica management

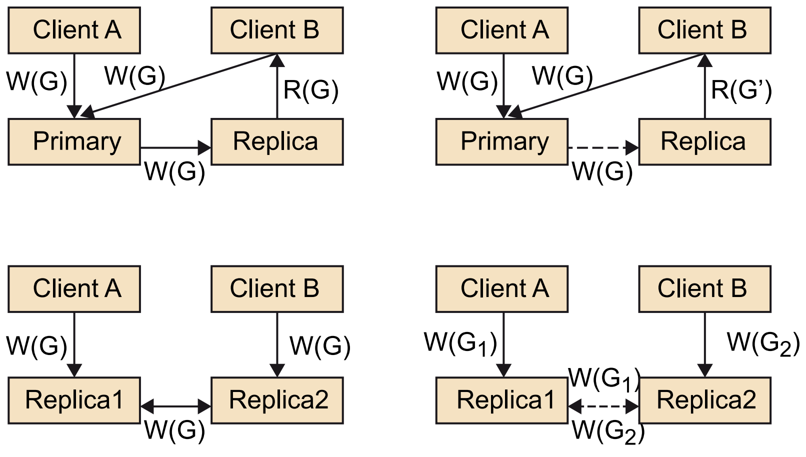


Figure 53 depicts all four possibilities resulting from the combination of two different dimensions to consider in this problem: We can maintain consistency synchronously (left hand side diagrams) or asynchronously (right hand side diagrams); and we can maintain a primary copy (top diagrams) or not (bottom diagrams).

Synchronous/ asynchronous operations

We say that an operation is synchronous, when it is completed between the user request and the confirmation from the system. Thus, an operation is said to be asynchronous when, even though it was confirmed by the system, its execution did not really finish at that moment.

- The left-top diagram shows that we have a primary copy with synchronously maintained replicas. In this case we can read from any copy, but have to modify the primary one (which can be a performance bottleneck). No inconsistencies can be generated, because changes are immediately propagated to the replicas before committing the operation. If the propagation is not possible (for example, because some node containing a copy is not reachable at this time), the operation is rejected by the system.

- The right-top diagram shows that we have a primary copy with asynchronously maintained replicas. We can read from any copy, but have to modify the primary one. The advantage of this approach is that we can optimize propagation of changes (i.e., we can confirm the operation, without the changes being propagated to all copies). The propagation of changes to the replicas can wait for the appropriate time. However, the primary node bottleneck remains and in the interval between the modification of the primary copy and the propagation of it to all replicas, some clients could get inconsistent results (i.e., two users would get a different value if their queries are answered retrieving data from different nodes).

- The left-bottom diagram shows that we do not have any primary copy, and all of them are synchronously modified. Now, modifications can be done at any replica (because all are actually modified at once) and no inconsistencies are obtained on reading, because all contain the same contents at any moment. Thus, users' queries can be answered by accessing the closest replica. The problem is that no modification can be committed until all replicas confirm they did the changes.

- Finally, right-bottom diagram shows that we do not have any primary copy and replicas are asynchronously maintained. Therefore, all bottlenecks disappear, because operations are always confirmed locally, but clients can get inconsistent results if two of them modify the same granule at the same time. When detected during the change propagation process, potential inconsistencies have to be reconciled (i.e., some copy must prevail over the others).

In highly distributed systems (thousands of nodes spread around the world), replica management becomes crucial. Moreover, the cost of transferring data and the chances of communications failure are really high, when we are considering Wide Area Networks (WANs). In 2000, the following conjecture (two years later was proved to be true) was proposed:

> "We can only achieve two of Consistency, system Availability, and tolerance to network Partition."
>
> CAP theorem, Eric Brewer

As above, consistency is used in this theorem in the sense that all users see the same data at the same time (the access point to the DDBMS does not matter). Availability means that if some nodes fail or are not reachable, the others can keep on working. Finally, tolerance to network partition means that even if some messages are lost, the system can continue operating. In larger distributed-scale system (i.e., thousands of computers working collaboratively), network partitions are given. Thus, we must choose between consistency and

Messages

As we explained in Section 4, messages between nodes are used for request operations, confirm their execution, lock data, etc.

availability: Either we have an always-consistent system that becomes temporally unavailable, or an always-available system that temporally shows some inconsistencies. This corresponds to the idea that consistency is not compatible with highly distributed data management, which gave rise to the NOSQL movement, that among other things proposes relaxing ACID properties.

In view of this, we have to generalize replica management to any number of replicas (not only two). Let us define N as the number of existing copies, W the number of copies that have to be written before the commit of the operation, and R the minimum number of copies that need to be read before giving the results to the client. These are parameters whose values have to be fixed by the database administrator, to accomplish the application requirements. We are especially interested in the case where users' modifications do not necessarily write all copies before the end of the operation (i.e., $W \neq N$). If we sychronously write all copies, reading only one is enough to get the right value. However, if we leave some copies temporally out of date, we need to read more than one copy to increase the chances of retrieving one with the right value.

Thus, in case of replication of data along different nodes, we can adopt different policies, that will affect the consistency of data:

Note

Strong consistency is achieved by synchronous replica management, while asynchronous replica management results in weak consistency.

- **Strong consistency**: After a modification has been committed, any user will read the same contents. Strong consistency can only be guaranteed if $W + R > N$. This means that the user will check data from at least one node containing the right value. Thus, if discrepancies are detected among the R replicas retrieved, more replicas are accessed until we can decide which one is correct. You should notice that by potentially accessing all nodes, we will always know that the value repeated W times is the right one.
  – Read One/Write All (ROWA): Users read any copy, and have to synchronously write all of them before committing (i.e., $R = 1$ and $W = N$). Read operations are prioritized over write ones, which will result to be really slow and with a low probability of succeeding in a highly distributed system prone to network partition.
  – Fault tolerant system: This corresponds to the configuration $N = 3$; $R = 2$; $W = 2$. This means that we keep three copies of each granule (in different nodes of the distributed system), and before reading or writing, users need to access two of these copies. Note that with this configuration the system is up even in the event of failure of one node.

- **Weak consistency**: The system does not guarantee that subsequent read will return the updated value. This happens when $W + R \leq N$. With massive replication for read scaling, some systems are configured with $R = 1$ and a relatively small value for W (e.g., 1 or 2). The time elapsed between the commit of the modification and the end of its propagation to enough copies is the inconsistency window.

- Primary copy: Users can read any copy, and there is one master copy they have to write before committing (i.e., $R = 1$ and $W = 1$). Changes in the master copy are eventually propagated to all copies.
- Eventually consistent: Users read some copies, and write also some copies before committing. All copies are eventually synchronized to converge to the same values. You should notice that in this configuration, conflicting modifications can be committed in different nodes. If this is the case, they have to be concilied as explained in next section.

5.1. BASE Transactions

BASE stands for Basically Available, Soft state, Eventual consistency; and appeared to oppose ACID transactions. If we write enough copies before the end of the operation, no data will be lost (the number of writes needed is $W \geq (N + 1)/2$). In this case, by a simple system we can decide which is the right value (it is guaranteed that half of the nodes plus one will always agree on a value). Nevertheless if we do not guarantee that we are able to write half of the copies, it may happen that two users confirm and their sets of written copies do not overlap. In this case, some changes would be lost during the synchronization phase if the two modifications are not compatible.

In this replica management approach (also known are optimistic replication), some modifications may be lost and users may access inconsistent data. It assumes that different granules are absolutely independent and that only the final state of the copies matters. In the presence of message loss, the system commits the changes, but keeps on sending the lost message until it is acknowledged by the corresponding node. Therefore, sooner or later, the system will detect potential inconsistencies (on synchronizing replicas) and deal with them (by discarding some modifications, if necessary).

Different granules are considered independent and it is only ensured that all replicas would eventually converge to the same value (if we would not get more modifications for some time).

In order to formalize the loss of modifications to be able to converge in the values of the replicas, two schedules are defined to be state-equivalent when starting at the same initial state, they produce the same final state (i.e., the same data).

Being $S^x{}_N$ the schedule of granule x at node N, it describes the operations performed over x at N. An element w_i in $S^x{}_N$ represents the execution of an update operation to x, while $\overline{w_i}$ shows that the operation was received at N but not executed (i.e., ignored) due to conflict resolution.

Note

In the classical serializability theory, two schedules are equivalent if besides producing the same final state, no modification is lost and the data seen by all transactions coincide in both cases.

For any node N containing granule x, if there is another node N' with a replica, we define the *committed prefix* as the prefix of $S^x{}_N$ state-equivalent to a prefix of $S^x{}_{N'}$. This always exists, since in the worse case, it will be the empty set of operations (i.e., an empty granule). For example, if $S_1{}^x = w_1$ and $S_2{}^x = w_2$, then $S_1{}^x \cap S_2{}^x = \varnothing$ would be their committed prefix and both would converge to $S_1{}^x = \overline{w_1}w_2$ and $S_2{}^x = w_2\overline{w_1}$. Three statements must be true for this to work properly:

1) The committed prefix grows monotonically (meaning that the set of operations and the relative order remain unchanged). Changes overwrite what was in the granule, if necessary.

2) For every operation w_i, either w_i or $\overline{w_i}$ eventually appears in the committed prefix (meaning that either sooner or later, any modification is taken into account or discarded everywhere), but not both, and not more than once.

3) If an operation w_i appears in the committed prefix, the state of the object in the state immediately before satisfies all the preconditions of the operation that allow to execute it.

Facebook

Systems like Facebook use this mechanism underneath. That's why some times posts are lost.

Let us suppose that we have $N = 2$; $W = 1$; $R = 1$. In this case, it is enough to read or write one out of the two copies of a granule to confirm the execution of the operation. We have two nodes $Node_1$ and $Node_2$ holding the copies of granule G. The committed prefix is the operation $G := 0$, because that is the value of the granule at both copies. Now, there is a network partition between $Node_1$ and $Node_2$. During this network partition, $User_1$ executes operation $G := 1$ at $Node_1$, and $User_2$ executes operation $G := 2$ at $Node_2$. Since $W = 1$, both users receive the confirmation of the execution even though the system is not able to check all the replicas. Eventually, the network problem is solved, and the system tries to synchronize both copies of G. $Node_1$ sends a message to $Node_2$ communicating a change from 0 to 1, while $Node_2$ sends a message to $Node_1$ communicating a change from 0 to 2. It is at this point that the problem is detected. To solve it, one of the two operations is ignored at both nodes. For example, the system could decide to ignore the operation issued by $User_1$, and thus both copies of G would have value 2.

Summary

In this module, we have seen that there are many concurrency control mechanisms. Some are more optimistic than others in the sense that they assume that interferences will not appear. Therefore, the more appropriate choice depends of the probability of having interferences. If many interferences between users are expected, locking is the best option. However, if few interferences may appear or rolling back transaction is cheap (e.g., read-only transactions), other techniques like timestamping are better suited. DBMSs like Oracle use multi-version for read-only transactions and locking for read-write.

Moreover, distributed environments become more and more relevant every day. Centralized concurrency control (either locking or timestamping) can be adapted to work in a distributed environment. However, timestamping is easier and more appropriate in most distributed environments.

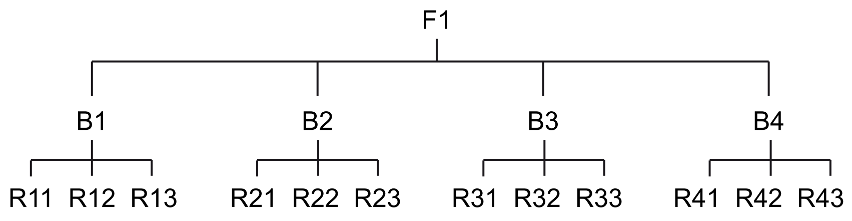
We have also seen that another important issue that becomes relevant in a distributed environment from the point of view of concurrency control is replica management. Replication is really important in highly distributed systems, and sacrificing consistency of the replicas has lately become very popular, giving rise to eventually consistent transactions.

Self-evaluation

1. Let us suppose that we have a DBMS without any kind of concurrency control mechanism, with the following schedule.

T_1	T_2	T_3
R(R21)		
	R(R21)	
		RU(R11)
		W(R11)
	R(R23)	
		RU(R22)
		W(R22)
R(R33)		
		RU(R43)
		W(R43)
		COMMIT
	COMMIT	
COMMIT		

Now consider a DBMS whose locking manager uses a multi-granule technique like the one you studied in this module, consider that it works at the levels of File, Block and Record with the following tree of granules:



T_1 locks blocks, while T_2 and T_3 lock records. Read-for-update (*RU*) operations generate exclusive locks, and read operations (*R*) generate shared locks. Let us suppose also that SIX mode is never used, and the transactions use Strict-2PL. How would this change the schedule (explicit the lock operations)?

2. Let us suppose that we have a DBMS without any kind of concurrency control mechanism, with the following schedule.

#Acc	T_1	T_2	T_3
10	BoT		
20	R(D)		
30	R(A)		
40		BoT	
50		R(E)	
60		W(E)	
70		R(C)	
80		W(C)	
90	W(A)		
100	R(B)		
110			BoT
120			R(F)
130			W(F)
140			R(E)
150	W(B)		
160		R(B)	
170			R(A)
180		W(E)	
190		COMMIT	
200	COMMIT		
210			COMMIT

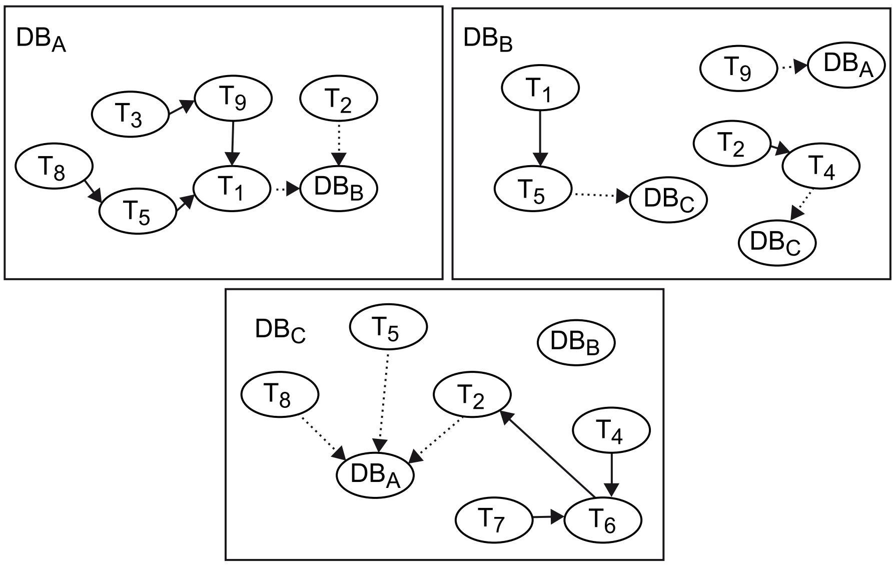
Let us consider, now, a concurrency control mechanism based on pure timestamping is put in place. How would this affect the schedule, considering that for all transaction T_i, $TS(T_i) = i$? Would any transaction be aborted?

3. Let us suppose that we have a DBMS without any kind of concurrency control mechanism, with the following schedule.

#Acc	T_1	T_2	T_3
10		BoT	
20	BoT		
30			BoT
40			R(A)
50			W(A)
60		R(B)	
70		W(B)	
80		R(C)	
90	R(D)		
100	W(D)		
110	R(E)		
120		R(F)	
130			R(B)
140		R(A)	
150	R(A)		
160	COMMIT		
170		COMMIT	
180			COMMIT

Let us consider, now, a multi-version concurrency control mechanism based on timestamping is put in place. How would this affect the schedule, considering that for all transaction T_i, $TS\ (T_i) = i$? Would any transaction be aborted? Which will be the versions of every granule (give the *RTS* and *WTS* of each)?

4. Let us suppose that we have a DDB in three sites (i.e., DB_A, DB_B, and DB_C). The concurrency control mechanism is based on locking, and we have a Wait-For graph at every node of the DDBMS (they are shown in the figure below). Use the algorithm explained in this module to find the global deadlocks, considering that for all transaction T_i, $TS\ (T_i) = i$.


DB_A
T3
T9
T2
T8
T1
DB_B
T5
DB_B
T1
T9
DB_A
T2
T4
T5
DB_C
DB_C
DB_C
T5
DB_B
T8
T2
T4
DB_A
T7
T6

Answer key

Self-evaluation

1.

T_1	T_2	T_3
Lock(B2,S)		
R(R21)		
	Lock(R21,S)	
	R(R21)	
		Lock(R11,X)
		RU(R11)
		W(R11)
	Lock(R23,S)	
	R(R23)	
		Lock(R22,X)
Lock(B3,S)		🟩
R(R33)		🟩
COMMIT	COMMIT	🟩
		RU(R22)
		W(R22)
		Lock(R43,X)
		RU(R43)
		W(R)
		COMMIT

2. The only change in the schedule is that at time 180, T_2 aborts (because of overwriting the value of granule *E* that T_3 read), and consequently T_3 also has to be aborted (because it read a value of *E* that never existed).

3. There is not any change in the schedule and no transaction is aborted. After time 180, the versions are as follows:

Granule	*RTS*	*WTS*
A_0	3	0
B_0	1	0
C_0	1	0
D_0	2	0
E_0	2	0
F_0	1	0
A_1	3	3
B_1	3	1
D_1	2	2

4. We have two different cycles: T_5–T_1 that would be detected at site DB_A; and T_2–T_4–T_6 that would be detected at site DB_C.

Glossary

ACID This acronym defines a kind of transactions that guarantee the following properties: Atomicity, Consistency, Isolation and Durability.

BoT Beginning of transaction

Conflicting transactions Two transactions have a conflict when they access the very same granule and at least one of them modifies it (rougthly speaking, it means that the relative execution order between them affects the result of the execution).

Contention Reduction of the throughput of the system due to mutual exclusion.

DB Database

DDB Distributed Database

DDBMS Distributed Database Management System

DBMS Database Management System

DDL Data Definition Language, refers to SQL statements that affect the schema of the tables (i.e., `CREATE`, `DROP`, etc.).

DML Data Manipulation Language, refers to SQL statements that affect the data in the tables (i.e., mainly `INSERT`, `UPDATE`, and `DELETE`).

EoT End of Transaction

Granule Unit of data the DBMS is locking (i.e., records, blocks, files, etc.).

NOSQL Not Only SQL

Read Set (RS) Set of tuples read by a transaction.

Read Timestamp (*RTS*) Timestamp of the last transaction that read a granule.

Recoverability Area of study dealing with the behavior of a DBMS in case of power or hardware failure.

SQL Structured Query Language

Transaction Atomic execution unit of a DBMS

Wait-For graph Directed graph where each node represents a transaction and each edge shows that a transaction is waiting for the other to free some granule.

Write Set (WS) Set of tuples written by a transaction.

Write Timestamp (*WTS*) Timestamp of the last transaction that wrote a granule.

2PL Two Phase Locking protocol

2PC Two Phase Commit protocol

Bibliography

Abiteboul, S.; Manolescu, I.; Rigaux, P.; Rousset, M.-C.; Senellart, P. (2011). *Web Data Management and Distribution*. Cambridge Press.

Bernstein, P. A.; Hedzilacos, V.; Goodman, N. (1987). *Concurrency control and Recovery mechanisms in database systems*. Addison-Wesley.

Liu, L.; Özsu, M. T. (Eds.) (2009). Encyclopedia of Database Systems.Springer.

Melton, J.; Simon, A. R. (2002). *SQL 1999*. Morgan Kaufmann.

Özsu, T. M.; Valduriez, P. (2011). *Principles of Distributed Database Systems*. 3rd Edition, Prentice Hall.

Ramamritham, K.; Chrysanthis, P. K. (1996). *Advances in Concurrency Control and Transaction Processing-Executive Briefing*. IEEE.

Ramakrishnan, R.; Gehrke, J. (2003). *Database Management Systems*. 3rd Edition, McGraw-Hill.

Shasha, D.; Bonnet, P. (2003). *Database Tuning*. Elsevier.