Lock-II

Hi! In this chapter I am going to discuss with you about Locks in DBMS in great detail.

As you know now that Concurrency control and locking is the mechanism used by DBMSs for the sharing of data. Atomicity, consistency, and isolation are achieved through concurrency control and locking.

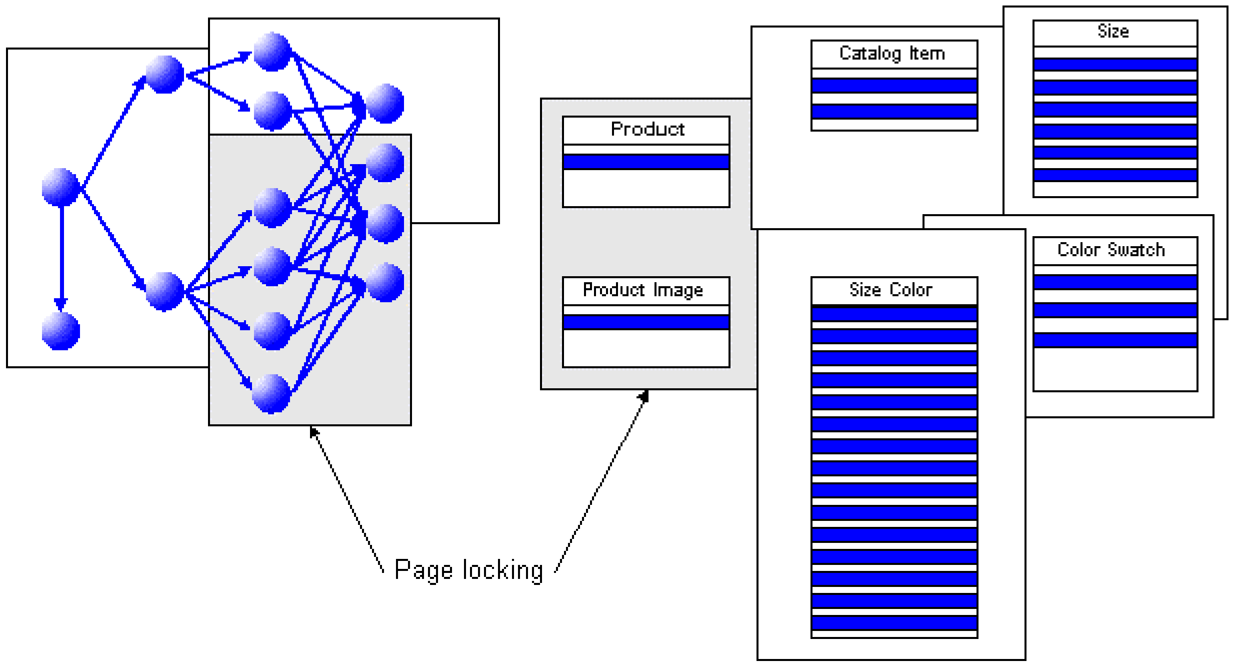
When many people may be reading the same data item at the same time, it is usually necessary to ensure that only one application at a time can change a data item. Locking is a way to do this. Because of locking, all changes to a particular data item will be made in the correct order in a transaction.

The amount of data that can be locked with the single instance or groups of instances defines the granularity of the lock. The types of granularity are illustrated here are:

- Page locking
- Cluster locking
- Class or table locking
- Object or instance locking

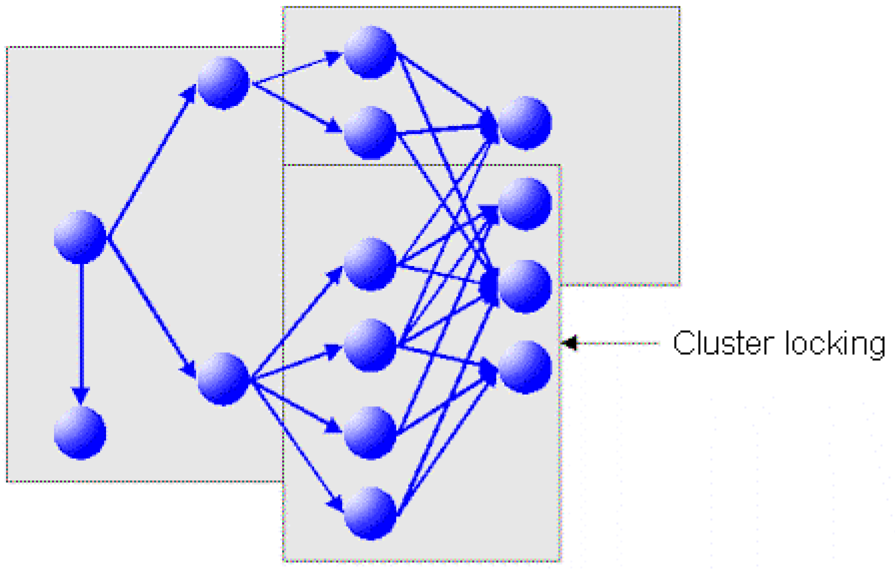
Page locking

Page locking (or page-level locking) concurrency control is shown in the figure below. In this situation, all the data on a specific page are locked. A page is a common unit of storage in computer systems and is used by all types of DBMSs. In this figure, each rectangle represents a page. Locking for objects is on the left and page locking for relational tuples is on the right. If the concept of pages is new to you, just think of a page as a unit of space on the disk where multiple data instances are stored.



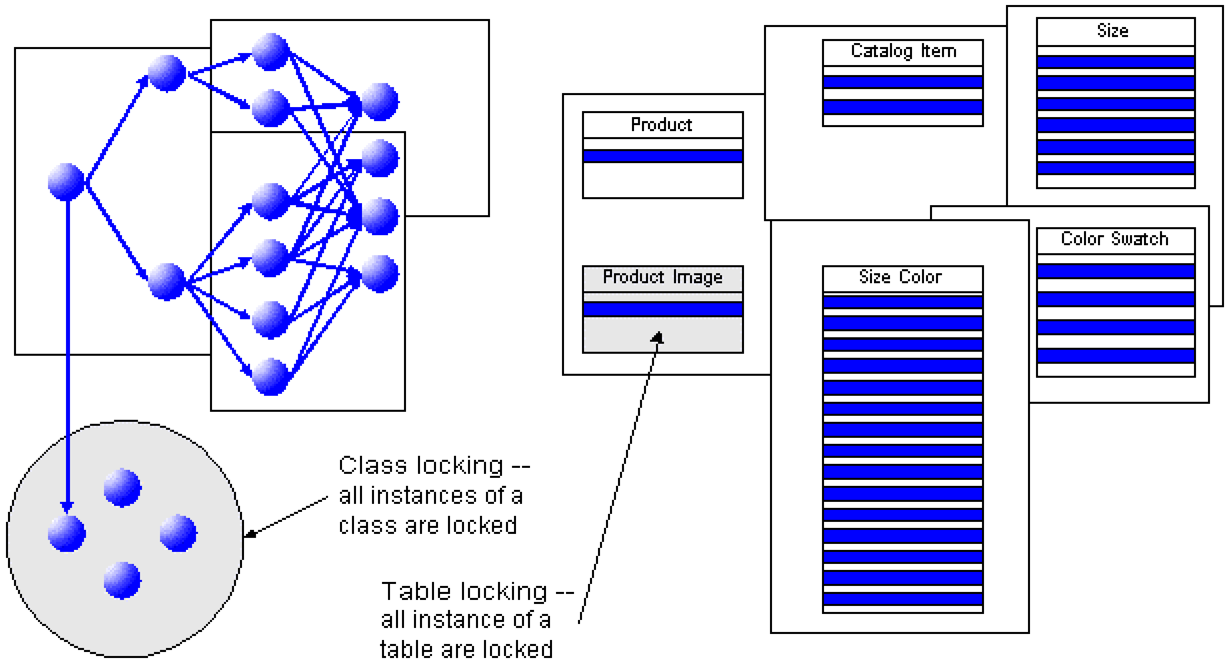
Cluster locking

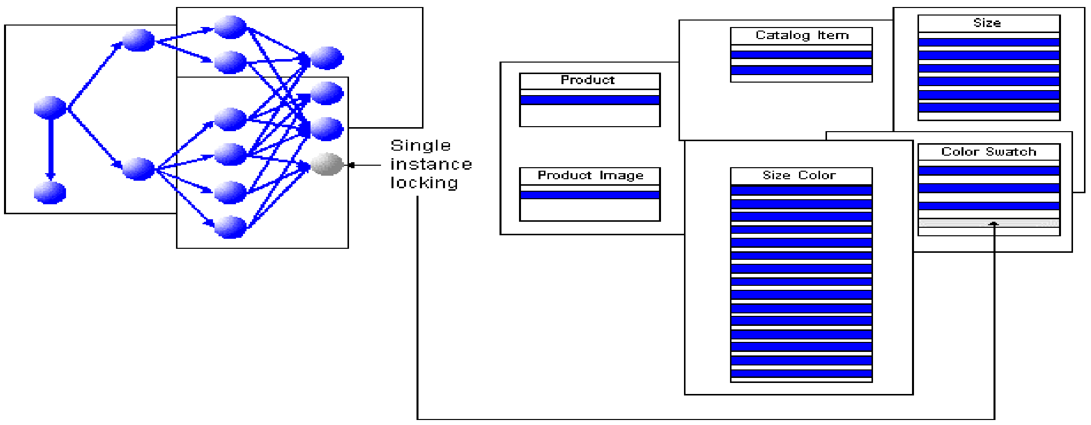
Cluster locking or container locking for concurrency control is illustrated in the figure below. In this form of locking, all data clustered together (on a page or multiple pages) will be locked simultaneously. This applies only to clusters of objects in ODBMSs. Note that in this example, the cluster of objects spans portions of three pages.



Class or table locking

Class or table locking means that all instances of either a class or table are locked, as is illustrated below. This shows one form of concurrency control. Note the circle at the lower left. It represents all instances of a class, regardless of the page where they are stored.


Product
Product Image
Catalog Item
Size
Color Swatch
Size Color
Class locking -- all instances of a class are locked
Table locking -- all instance of a table are locked



Two Phase Locking

A two-phase locking (2PL) scheme is a locking scheme in which a transaction cannot request a new lock after releasing a lock. Two phases locking therefore involves two phases:

- Growing Phase (Locking Phase) - When locks are acquired and none released.
- Shrinking Phase (Unlocking Phase) - When locks are released and none acquired.

The attraction of the two-phase algorithm derives from a theorem which proves that the two-phase locking algorithm always leads to serializable schedules. This is a sufficient condition for Serializability although it is not necessary.

Timestamp Ordering Protocol

In the previous chapter of locking we have studied about the 2 phase locking system. Here I would like to discuss with you certain other concurrency control schemes and protocols.

The timestamp method for concurrency control does not need any locks and therefore there are no deadlocks. Locking methods generally prevent conflicts by making transaction to walk. Timestamp methods do not make the transactions wait. Transactions involved in a conflict are simply rolled back and restarted. A timestamp is a unique identifier created by the DBMS that indicates the relative starting time of a transaction. Timestamps are generated either using the system clock (generating a timestamp when the transaction starts to execute) or by incrementing a logical counter every time a new transaction starts.

Time stamping is the concurrency control protocol in which the fundamentals goal is to order transactions globally in such away that older transactions get priority in the event of a conflict. In the Timestamping method, if a transaction attempts to read or write a data item, then a read or write operation is allowed only if the last update on that data item was carried out by an older transaction. Otherwise the transaction requesting the read or write is restarted and given a new timestamp to prevent it from continually aborting and restarting. If the restarted transaction is not allowed a new timestamp and is allowed a new timestamp and is allowed to retain the old timestamp, it will never be allowed to perform the read or write, because by that some other transaction which has a newer timestamp than the restarted transaction might not be to commit due to younger transactions having already committed.

In addition to the timestamp for the transactions, data items are also assigned timestamps. Each data item contains a read-timestamp and write-timestamp. The read-timestamp contains the timestamp of the last transaction that read the item and the write-timestamp contains the timestamp of the last transaction that updated the item. For a transaction T the timestamp ordering protocol works as follows:

- Transactions T requests to read the data item 'X' that has already been updated by a younger (later or one with a greater timestamp) transaction. This means that an earlier transactions is trying to read a data item that has been updated by a later transaction T is too late to read the previous outdated value and any other values it has acquired are likely to be inconsistent with the updated value of the data item. In this situation, the transaction T is aborted and restarted with a new timestamp.

- In all other cases, the transaction is allowed to proceed with the read operation. The read-timestamp of the data item is updated with the timestamp of transaction T.

- Transaction t requests to write (update) the data item ‘X’ that has already been read by a younger (later or one with the greater timestamp) transaction. This means that the younger transaction is already using the current value of the data item and it would be an error to update it now. This situation occurs when a transaction is late in performing the write and a younger transaction has already read the old value or written a new one. In this case the transaction T is aborted and is restarted with a new timestamp.

- Transaction T asks to write the data item ‘X’ that has already been written by a younger transaction. This means that the transaction T is attempting to write an old or obsolete value of the data item. In this case also the transaction T is aborted and is restarted with a new timestamp.

- In all other cases the transaction T is allowed to proceed and the write-timestamp of the data item is updated with the timestamp of transaction T.

The above scheme is called basic timestamp ordering. This scheme guarantees that the transactions are conflict serializable and the results are equivalent to a serial schedule in which the transactions are executed in chronological order by the timestamps. In other words, the results of a basic timestamps ordering scheme will be as same as when all the transactions were executed one after another without any interleaving.

One of the problems with basic timestamp ordering is that it does not guarantee recoverable schedules. A modification to the basic timestamp ordering protocol that relaxes the conflict Serializability can be used to provide greater concurrency by rejecting obsolete write operations. This extension is known as Thomas’s write rule. Thomas’s write rule modifies the checks for a write operation by transaction T as follows.

- When the transaction T requests to write the data item 'X' whose values has already been read by a younger transaction. This means that the order transaction (transaction T) is writing an obsolete value to the data item. In this case the write operation is ignored and the transaction (transaction T) is allowed to continue as if the write were performed. This principle is called the 'ignore obsolete write rule'. This rule allows for greater concurrency.
- In all other cases the transactions T is allowed to proceed and the write-timestamp of transaction T.

Thus the use of Thomas's write rule allows us to generate schedules that would not have been possible under other concurrency protocols.

Time stamping Control-Contrast to 2PL

The two-phase locking technique relies on locking to ensure that each interleaved schedule executed is serializable. Now we discuss a technique that does not use locks and works quite well when level of contention between transactions running concurrently is not high. It is called Time stamping.

Timestamp techniques are based on assigning a unique *timestamp* (a number indicating the time of the start of the transaction) for each transaction at the start of the transaction and insisting that the schedule executed is always serializable to the serial schedule in the chronological order of the timestamps. This is in contrast to two-phase locking where any schedule that is equivalent to some serial schedule is acceptable.

Since the scheduler will only accept schedules that are serializable to the serial schedule in the chronological order of the timestamps, the scheduler must insist that in case of conflicts, the junior transaction must process information only after the older transaction has written them. The transaction with the smaller timestamp being the older transaction. For the scheduler to be able to carry this control, the data items also have read and write timestamps. The read timestamp of a data item X is the timestamp of the youngest transaction that has read it and the write timestamp is the timestamp of the youngest transaction that has written it. Let timestamp of a transaction be TS(T).

Consider a transaction T_1 with timestamp = 1100. Suppose the smallest unit of concurrency control is a relation and the transaction wishes to read or write some tuples from a relation named *R*. Let the read timestamp of *R* be *RT(R)* and write timestamp be *WT(R)*. The following situations may then arise:

1. T_1 wishes to read *R*. The read by T_1 will be allowed only if $WT(R) < TS(T_1)$ or $WT(R) < 1100$ that is the last write was by an older transaction. This condition ensures that data item read by a transaction has not been written by a younger transaction. If the write timestamp of *R* is larger than 1100 (that is, a younger transaction has written it), then the older transaction is rolled back and restarted with a new timestamp.
2. T_1 wishes to read *R*. The read by T_1 will be allowed if it satisfies the above condition even if $RT(R) > TS(T_1)$, that is the last read was by a younger transaction. Therefore if data item has been read by a younger transaction that is quite acceptable.
3. T_1 wishes to write some tuples of a relation *R*. The write will be allowed only if read timestamp $RT(R) < TS(T_1)$ or $RT(R) < 1100$, that is the last read of the transaction was by an older transaction and therefore no younger transaction has read the relation.
4. T_1 wishes to write some tuples of a relation *R*. The write need not be carried out if the above condition is met and if the last write was by a younger transaction, that is, $WT(R) > TS(T_1)$. The reason this write is not needed is that in this case the younger transaction has not read *R* since if it had, the older transaction T_1 would have been aborted.

If the data item has been read by a younger transaction, the older transaction is rolled back and restarted with a new timestamp. It is not necessary to check the write (??) since if a younger transaction has written the relation, the younger transaction would have read the data item and the older transaction has old data item and may be ignored. It is possible to ignore the write if the second condition is violated since the transaction is attempting to write obsolete data item. [expand!]

Let us now consider what happens when an attempt is made to implement the following schedule:

We assume transaction T_1 to be older and therefore $TS(T_1) < TS(T_2)$.

Read (A) by transaction T_1 is permitted since A has not been written by a younger transaction. Write (A) by transaction T_1 is also permitted since A has not been read or written by a younger transaction. Read (B) by transaction T_2 is permitted since *B* has not been written by a younger transaction. Read (B) by transaction T_1 however is not allowed since *B* has been read by (younger) transaction T_2 and therefore transaction T_1 is rolled back. [needs to be read again and fixed]

<u>Optimistic Control</u>

This approach is suitable for applications where the number of conflicts between the transactions is small. The technique is unsuitable for applications like the airline reservations system where write operations occur frequently at ``hot spots'', for example, counters or status of the flight. In this technique, transactions are allowed to execute unhindered and are validated only after they have reached their commit points. If the transaction validates, it commits, otherwise it is restarted. For example, if at the time of validation, it is discovered that the transaction wanting to commit had read data item that has already been written by another transaction, the transaction attempting to commit would be restarted. [Needs more]

These techniques are also called *validation* or *certification* techniques.

The optimistic control may be summarized as involving checking at the *end* of the execution and resolving conflicts by *rolling back*. It is possible to combine two or more control techniques -- for example, very heavily used items could be locked, others could follow optimistic control.

Review Questions

1. Explain the types of granularity?
2. Explain the timestamp ordering protocol?
3. Explain the Timestamping control?
4. Explain optimistic control?

References

Date, C, J, Introduction to Database Systems, 7th edition
Silbershatz, Korth, Sudarshan, Database System Concepts 4th Edition.