### *Why is concurrency control needed?*

### If transactions are executed *serially*, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable result may occur, such as:

1. ***The lost update problem:***A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
2. ***The dirty read problem:*** Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.
3. ***The incorrect summary problem:*** While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

Most high-performance transactional systems need to run transactions concurrently to meet their performance requirements. Thus, without concurrency control such systems can neither provide correct results nor maintain their databases consistent.

------------------------

Concurrency Problems

If locking is not available and several users access a database concurrently, problems may occur if their transactions use the same data at the same time. Concurrency problems include:

* Lost or buried updates.
* Uncommitted dependency ([dirty read](https://technet.microsoft.com/en-us/library/aa213029(v=sql.80).aspx#sql:dirty_read)).
* Inconsistent analysis ([nonrepeatable read](https://technet.microsoft.com/en-us/library/aa213029(v=sql.80).aspx#sql:nonrepeatable_read)).
* Phantom reads.

*Lost Updates*

Lost updates occur when two or more transactions select the same row and then update the row based on the value originally selected. Each transaction is unaware of other transactions. The last update overwrites updates made by the other transactions, which results in lost data.

For example, two editors make an electronic copy of the same document. Each editor changes the copy independently and then saves the changed copy, thereby overwriting the original document. The editor who saves the changed copy last overwrites changes made by the first editor. This problem could be avoided if the second editor could not make changes until the first editor had finished.

*Uncommitted Dependency (Dirty Read)*

Uncommitted dependency occurs when a second transaction selects a row that is being updated by another transaction. The second transaction is reading data that has not been committed yet and may be changed by the transaction updating the row.

For example, an editor is making changes to an electronic document. During the changes, a second editor takes a copy of the document that includes all the changes made so far, and distributes the document to the intended audience. The first editor then decides the changes made so far are wrong and removes the edits and saves the document. The distributed document contains edits that no longer exist, and should be treated as if they never existed. This problem could be avoided if no one could read the changed document until the first editor determined that the changes were final.

*Inconsistent Analysis (Nonrepeatable Read)*

Inconsistent analysis occurs when a second transaction accesses the same row several times and reads different data each time. Inconsistent analysis is similar to uncommitted dependency in that another transaction is changing the data that a second transaction is reading. However, in inconsistent analysis, the data read by the second transaction was committed by the transaction that made the change. Also, inconsistent analysis involves multiple reads (two or more) of the same row and each time the information is changed by another transaction; thus, the term nonrepeatable read.

For example, an editor reads the same document twice, but between each reading, the writer rewrites the document. When the editor reads the document for the second time, it has changed. The original read was not repeatable. This problem could be avoided if the editor could read the document only after the writer has finished writing it.

*Phantom Reads*

Phantom reads occur when an insert or delete action is performed against a row that belongs to a range of rows being read by a transaction. The transaction's first read of the range of rows shows a row that no longer exists in the second or succeeding read, as a result of a deletion by a different transaction. Similarly, as the result of an insert by a different transaction, the transaction's second or succeeding read shows a row that did not exist in the original read.

For example, an editor makes changes to a document submitted by a writer, but when the changes are incorporated into the master copy of the document by the production department, they find that new unedited material has been added to the document by the author. This problem could be avoided if no one could add new material to the document until the editor and production department finish working with the original document.

--------------------

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Concurrency Problems  |  |  | | --- | --- | |  | [Lost Updates](https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/transmanagement/concurrency.htm#Lost Updates) | | [Uncommitted Data](https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/transmanagement/concurrency.htm#Uncommitted Data) | | [Inconsistent Retrievals](https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/transmanagement/concurrency.htm#Inconsistent Retrieval) | | [The Scheduler](https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/transmanagement/concurrency.htm#The Scheduler) |   The uncontrolled execution of concurrent transactions in a multi-user environment can lead to various problems.  The three main problems and examples of how they can occur are listed below:  **Lost Updates [top of page](https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/transmanagement/concurrency.htm#top)** This problem occurs when two transactions, accessing the same data items, have their operations interleaved in such a way, that one transaction will access the data before the other has applied any updates.   |  |  |  |  | | --- | --- | --- | --- | | **Transaction1** | **Transaction2** | **Operation** | **Data Value** | | Read Flight Information |  |  | seats = 15 | |  | Read Flight Information |  | seats = 15 | |  | Book 2 seats | seats = seats -2 |  | | Book 1 seat |  | seats = seats -1 |  | |  | Write seats |  | seats = 13 | | Write seats |  |  | seats = 14 |   In this simplified example, you can see that the operations are interleaved in such a way that Transaction 2 had access to the data before Transaction 1 could reduce the seats by 1.  Transaction 2's operation of reducing the seats by 2 has been overwritten, resulting in an incorrect value of available seats.  This violates the **Serializability** property which requires that the results of interleaving must leave the database with the same results as serial processing.  It also violates the **Isolation**property of allowing a transaction to complete without interference from another.  **Uncommitted Data [top of page](https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/transmanagement/concurrency.htm#top)** This problem occurs when one transaction updates a data item, but has not yet committed the data permanently to the database.  Because of  failure, the transaction is rolled back and the data item is returned to its previous value.  A second transaction accesses the updated data item before it is returned to its original value.   |  |  |  |  | | --- | --- | --- | --- | | **Transaction1** | **Transaction2** | **Operation** | **Data Value** | | Read Flight Information |  |  | seats = 15 | | Book 1 seat |  | seats = seats -1 |  | | Write seats |  |  | seats = 14 (Uncommitted) | |  | Read Flight Information |  | seats = 14 | |  | Book 2 seats | seats = seats -2 | seats = 12 | |  | Write seats |  | seats = 12 | | Rollback |  |  | seats = 15 |   In this example, you can see that Transaction 2 has access to the updated data of Transaction 1 before the changes were committed permanently to the database.  This violates the **Isolation**property of allowing a transaction to complete without interference from another.  **Inconsistent Retrievals [top of page](https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/transmanagement/concurrency.htm#top)** This problem occurs when one transaction is calculating summary functions on particular data items while other transactions are updating those data items.  The transaction performing the calculations may read some data items before they are updated and others after.  Assume Flight F106 has 12 seats available and Flight F113 has 4 seats available giving a total of 16 available seats.  If Transaction 1 were to update Flight F106 by booking 1 seat and Transaction 2 were to update Flight F113 by booking 2 seats, a calculation of the total seats available would yield a result of 13 seats.  The following example illustrates what could happen when other transactions are permitted to update while this calculation is occurring.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Transaction1** | **Transaction2** | **Transaction3** | **Operation** | **Data Value** | |  |  | Read seats from flight F106 |  | F106 seats=12 | |  |  | Add F106 seats to Total Seats | Total seats = Total seats+12 | Total Seats=12 | | Read Flight F106 Information |  |  |  | F106 seats= 12 | | Book 1 seat |  |  | F106 seats = F106 seats-1 |  | | Write seats |  |  |  | F106 seats= 11 | |  | Read Flight F113 Information |  |  | F113 seats= 4 | |  | Book 2 seats |  | F113 seats = F113 seats-2 |  | |  | Write seats |  |  | F113 seats= 2 | |  |  | Read seats from flight F113 |  | F113 Seats= 2 | |  |  | Add F113 seats to Total Seats | Total seats = Total seats+2 | Total Seats= 14 |   This also violates the **isolation**property of allowing a transaction to complete without interference from another.  **The Scheduler [top of page](https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/transmanagement/concurrency.htm#top)**  Most concurrency problems in a multi-user environment occur because the order of operations is incorrect.  Interleaving operations must be sequenced correctly to ensure **isolation** and **serializability**.  Determining the correct order of interleaved operations is the task of **the scheduler**.  The scheduler determines the order by employing the concurrency control techniques discussed in the next three topics.  **[top of page](https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/transmanagement/concurrency.htm#top)** |

------------------------------------------------------------

**NEED FOR CONCURRENCY CONTROL**

**Concurrency Control**

**Process of managing simultaneous operations on the database without having them interfere with one another.**

**• Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.**

**• Although two transactions may be correct in themselves, interleaving of operations may produce**

**an incorrect result.**

**Need**

**Several problems can occur when concurrent transactions execute in an uncontrolled manner.**

**1) The Lost Update Problem**

**This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.**

**Successfully completed update is overridden by another user.**

**Example:**

**• T1 withdraws £10 from an account with balx, initially £100.**

**• T2 deposits £100 into same account.**

**• Serially, final balance would be £190.**

**Loss of T2's update!!**

**This can be avoided by preventing T1 from reading balx until after update.**

**The Temporary Update (or Dirty Read) Problem**

**This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.**

**Occurs when one transaction can see intermediate results of another transaction before it has committed.**

**Example:**

**• T4 updates balx to £200 but it aborts, so balx should be back at original value of £100.**

**• T3 has read new value of balx (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.**

**Problem avoided by preventing T3 from reading balx until after T4 commits or aborts.**

**The Incorrect Summary Problem**

**If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.**

**Occurs when transaction reads several values but second transaction updates some of them during execution of first.**

**Example:**

**• T6 is totaling balances of account x (£100), account y (£50), and account z (£25).**

**• Meantime, T5 has transferred £10 from balx to balz, so T6 now has wrong result (£10 too high).**

**Problem avoided by preventing T6 from reading balx and balz until after T5 completed updates.**

**---------------------------------------------------**

Process of managing simultaneous execution of transactions in a shared database, to ensure the serializability of transactions, is known as concurrency control.

***Why we need Concurrency Control***

*Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems:*

* + - *Lost Updates.*
    - *Uncommitted Data.*
    - *Inconsistent retrievals.*

***When we need Concurrency Control***

Concurrent access to data is desirable when:

1. The amount of data is sufficiently great that at any given time only fraction of the data can be in primary memory & rest should be swapped from secondary memory as needed.
2. Even if the entire database can be present in primary memory, there may be multiple processes.

***Concurrency Control Techniques***

• Pessimistic concurrency control

– Locking

• Optimistic concurrency control

***Pessimistic Concurrency Control***

* Pessimistic Concurrency Control assumes that conflicts will happen
* Pessimistic Concurrency Control techniques detect conflicts as soon as they occur and resolve them using blocking***.***

***Locking***

* Locking is “pessimistic” because it assumes that conflicts will happen.
* The concept of locking data items is one of the main techniques used for controlling the concurrent execution of transactions.
* A lock is a variable associated with a data item in the database. Generally there is a lock for each data item in the database.
* A lock describes the status of the data item with respect to possible operations that can be applied to that item. It is used for synchronising the access by concurrent transactions to the database items.

– A transaction locks an object before using it

– When an object is locked by another transaction, the requesting transaction must wait

***Disadvantages of locking***

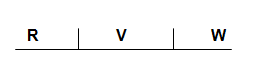
* Lock management overhead.
* Deadlock detection/resolution.
* Concurrency is significantly lowered, when congested nodes are locked.
* To allow a transaction to abort itself when mistakes occur, locks can’t be released until the end of transaction, thus currency is significantly lowered
* (Most Important) Conflicts are rare. (We might get better performance by not locking, and instead checking for conflicts at commit time.)

***Optimistic Concurrency Control***

* Optimistic Concurrency Control assumes that conflicts between transactions are rare.
* Does not require locking
* Transaction executed without restrictions
* Check for conflicts just before commit

***Phases for Optimistic Concurrency Control***

* + Read Phase
  + Validation Phase
  + Write Phase



***Read Phase***

* No global writes take place
* Whenever the first write to a given object is requested, a copy is made, and all subsequent writes are directed to the copy.
* When the transaction completes, it requests its validation and write phases.

***Write Phase***

* If the validation fails, the transaction will be backed up and started again as a new transaction
* If validation succeeds, then the transaction enters the write phase where locally written data are made global.

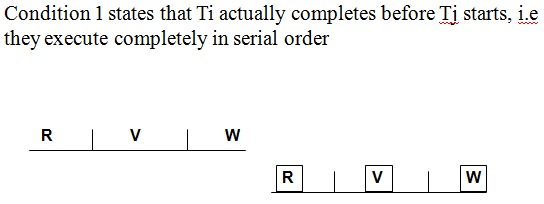
***Validation Phase***

* Checks are performed to ensure serializibility is not violated if the transaction updates are applied to the database.
* For read only validation consists of checks to ensure that the values read are the current values for the corresponding data items. If not interference has occurred, and the transaction is backed up and restarted.
* For a transaction that has updates, validation consists of determining whether the current transaction leaves the database in a consistent state, with serializibility maintained. If not, transaction is backed up and restarted.
* The scheduling of transactions is done by assigning transaction numbers to transactions
* Each Transaction Ti is explicitly assigned a unique integer transaction number t(i)
* There must exist a serially equivalend schedule in which transaction Ti comes before transaction Tj whenever t(i) < t(j).

To guarantee this numbering criteria one of the following three conditions must hold:

1. Ti completes its write phase before Tj starts its read phase.
2. The write set of Ti does not intersect the read set of Tj , and Ti complete its write phase before Tj starts its write phase.
3. The write set of Ti does not intersect the read set or write set of Tj , and Ti completes its read phase before Tj complete its read phase.

***Condition 1***

******

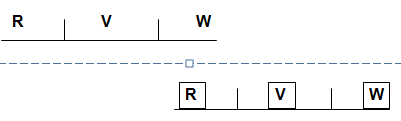
***Condition 2***

Condition 2 states that the writes of Ti do not affect the read phase of Tj, and that Ti finishes writing before Tj starts writing.

WS(Ti) disjoint from RS(Tj)

Ti does not overwrite Tj

Tj does not read any dirty data since Tj read data when Ti is still modifying it.



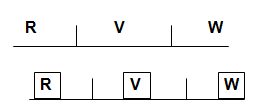
***Condition 3***

Condition 3 is similar to Condition 2 but does not require that Ti finish writing before Tj starts writing; it simply requires that Ti does not affect the read phase or the write phase of Tj

* + WS(Ti) disjoint from RS(Tj)

This condition allows Ti and Tj to write objects at the same time, but there is no overlapping for the sets written by these two transactions.

* + WS(Ti) disjoint from WS(Tj).



***Assigning Transaction Numbers***

* Transactions numbers should be assigned in order, since if Ti completes before Tj starts Ti < Tj.

Solution: Maintain a global counter, when transaction number is needed increment the counter and return the value.

* Transaction number must be assigned somewhere before validation, since validation require knowledge of the transaction number for the transaction being validated.

– Transaction numbers can be assigned at the beginning of the read phase. But this is not optimistic.

– Therefore these are usually assigned at the end of read phase

***Practical Considerations***

* Space for WriteSets: To validate, Tj must have WriteSets for all Ti where Ti < Tj and Ti was active when Tj began. There may be many such transactions, and we may run out of space.

Solution:

– Concurrency Control maintain some finite number of most recent write sets, where the number is large enough to validate almost all transactions

– If old write sets are unavailable the validation fails and the transaction is backed up.

* ***Starvation: What should be done when validation repeatedly fails ?***

***Solution:***

– If the concurrency control detects a starving transaction, it will be restarted, but without releasing the critical section semaphore, and transaction is run to the completion by write locking the database

***Serial Validation***

* Implementation of Condition number 1 and 2. Since Condition 3 is not used, last part of condition 3 implies that write phases must be serial (writes won’t be interleaved).

– Ti finishes writing before Tj starts reading (serial)

– WS(Ti) disjoint from RS(Tj), and Ti finish writing before Tj starts writing.

***IMPLEMENTATION:->***

* Place the assignment of a transaction number, validation and the subsequent write phase all in a critical section. (Since nothing else goes on concurrently, no need to check for Condition 3)

– Write sets of all transactions started form the beginning to the end of this transaction are checked to see whether they intersect with the read set of current transaction.

– If validation succeed write phase is committed, otherwise transaction is backed up.

* Optimization for Read-only Transactions:
  + Since there is no write phase we don’t need critical section

***PARALLEL VALIDATION***

* Parallel validation uses all 3 validation Conditions, thus allowing greater concurrency (For allowing interleaved writes).

– Maintain a set of transactions ids active for transactions that have completed their read phase but have not yet completed their write phase.

– During validation, in addition to the rules for serial validation, active set’s write set is checked for intersections with the read & write sets of the current transaction.

– If validation succeed write phase is committed, otherwise transaction is backed up.

***CONCLUSION:->***

* Optimistic Concurrency Control is superior to locking methods for systems where transaction conflict is highly unlikely, e.g query dominant systems.

– Avoids locking overhead

– Using parallel validation OCC can take full advantage of multiprocessor environment.