Concurrency control

- The workload of operational DBMSs is measured in tps, i.e., transactions per second
  - \( 10^{-3} \) for banking applications and flight reservations
- Concurrency control provides **concurrent access** to data
  - It increases DBMS efficiency by
    - maximizing the number of transactions per second (throughput)
    - minimizing response time

Elementary I/O operations

- Elementary operations are
  - Read of a single data object \( r(x) \)
  - Write of a single data object \( w(x) \)
- They may require reading from disk or writing to disk an entire page

Scheduler

- The **scheduler**
  - is a block of the concurrency control manager
  - is in charge of deciding if and when read/write requests can be satisfied
- The absence of a scheduler may cause correctness problems
  - also called anomalies

Lost update

- The **correct** value is \( x=4 \)
- The effect of transaction \( T_2 \) is **lost** because both transactions read the same initial value

```
Transaction T1
bot
r_1(x) x=2
x= x+1 x=3

Transaction T2
bot
r_2(x) x=2
x= x+1 x=3
w_2(x) x=3
commit
```
**Dirty read**

Transaction $T_1$  
- $r_1(x) \quad x=2$  
- $x \leftarrow x+1 \quad x=3$  
- $w_1(x) \quad x=3$

Transaction $T_2$  
- $r_2(x) \quad x=2$

Cascade rollback

Transaction $T_2$ reads the value of $X$ in an intermediate state which never becomes stable (permanent).

**Inconsistent read**

Transaction $T_1$  
- $r_1(x) \quad x=2$

Transaction $T_2$  
- $r_2(x) \quad x=3$
- $w_2(x) \quad x=3$

Commit

Commit

Transaction $T_1$ reads $x$ twice  
- $x$ has a different value each time

**Ghost update (a)**

Transaction $T_1$  
- $r_1(x) \quad x=400$
- $r_1(y) \quad y=300$

Transaction $T_2$  
- $r_2(y) \quad y=300$
- $y = y - 100 \quad y=200$
- $z = z + 100 \quad z=400$
- $w_2(y) \quad y=200$
- $w_2(z) \quad z=400$

Commit

Commit

Total = $x + y + z \quad total=1100$

The correct value is total = $400+200+400=1000$

**Ghost update (b)**  
- The insert operation is the ghost update  
- Problem  
- The data is not yet in the database before the insert

**Ghost update (b)**

Transaction $T_1$  
- read the salary of all employees in department $x$ and compute AVG salary

Transaction $T_2$  
- insert a new employee in department $x$ and compute AVG salary  
- read the salary of all employees in department $x$ and compute AVG salary

Commit
**Schedule**

- The *transaction* is a sequence of read and write operations characterized by the same TID (Transaction Identifier)
  \[ r_1(x) \ r_1(y) \ w_1(x) \ w_1(y) \]
- The *schedule* is a sequence of read/write operations presented by concurrent transactions
  \[ r_1(z) r_2(z) w_1(y) w_2(z) \]
  - Operations in the schedule appear in the arrival order of requests

**Scheduler**

- Concurrency control accepts or rejects schedules to avoid anomalies
- The scheduler has to accept or reject operation execution *without knowing the outcome* of the transactions
  - abort/commit

**Commit projection**

- Commit projection is a simplifying hypothesis
  - *The schedule only contains transactions performing commit*
- The dirty read anomaly is not addressed
- This hypothesis will be removed later

**Serial schedule**

- In a *serial schedule*, the actions of each transaction appear in sequence, without interleaved actions belonging to different transactions
- Example
  \[
  r_0(x) \ r_0(y) \ w_0(x) \ r_2(x) \ r_2(y) \ r_1(y) \ w_1(x) \ w_1(y)
  \]

**Serializable schedule**

- An arbitrary schedule \( S_i \) (commit projection) is correct when it yields the same result as an arbitrary serial schedule \( S_j \) of the same transactions
- \( S_i \) is *serializable*
  - \( S_i \) is equivalent to an arbitrary serial schedule of the same transactions
### Equivalence between schedules

- Different equivalence classes between two schedules
  - View equivalence
  - Conflict equivalence
  - 2 phase locking
  - Timestamp equivalence
- Each equivalence class
  - Detects a set of acceptable schedules
  - Is characterized by a different complexity in detecting equivalence

### View equivalence

- Definitions
  - `reads-from`
    - `r(x)` reads-from `w(x)` when
      - `w(x)` precedes `r(x)` and `i ≠ j`
      - There is no other `w(x)` between them
  - `final write`
    - `w(x)` is a final write if it is the last write of `x` appearing in the schedule
- Two schedules are **view equivalent** if they have
  - The same `reads-from` set
  - The same final write set

### View serializable schedule

- A schedule is **view serializable** if it is view equivalent to an arbitrary serial schedule of the same transactions
- VSR: schedules which are view serializable
- Example
  - `S_1 = w_0(x) r_1(x) r_2(x) w_2(x) w_1(x)`
  - `S_2 = w_0(x) r_1(x) r_2(x) w_2(x) w_2(x)`
- `S_1` is view serializable because it is view equivalent to `S_2`

### Lost update anomaly

- Transaction `T_1`
  - `bot`
  - `r_1(x)`
  - `x` = `x+1`
  - `w_1(x)`
  - `commit`

- Transaction `T_2`
  - `bot`
  - `r_2(x)`
  - `x` = `x+1`
  - `w_2(x)`
  - `commit`

- Corresponding schedule
  - `S = r_1(x) r_2(x) w_2(x) w_1(x)`
Inconsistent read anomaly

Transaction $T_1$
- $r_1(x)$
- $r_1(y)$
- $r_1(z)$
- commit

Transaction $T_2$
- $r_2(x)$
- $r_2(y)$
- $r_2(z)$
- $w_2(x)$
- $w_2(y)$
- $w_2(z)$
- commit

Corresponding schedule
$S = r_1(x) \ r_2(x) \ w_2(x) \ r_1(x)$

Is this schedule serializable?

Only two possible serial schedules

$S_1 = r_1(x) \ r_2(x) \ r_1(x) \ w_2(x)$

$S_2 = r_2(x) \ w_2(x) \ r_1(x) \ r_1(x)$

$S$ is not view equivalent to any serial schedule
- not serializable
- should be rejected

Ghost Update (a)

Transaction $T_1$
- $r_1(x)$
- $r_1(y)$
- $r_1(z)$
- total = $x + y + z$
- commit

Transaction $T_2$
- $r_2(x)$
- $r_2(y)$
- $r_2(z)$
- $y = y - 100$
- $z = z + 100$
- $w_2(y)$
- $w_2(z)$
- commit

$S = r_1(x) \ r_2(y) \ r_1(y) \ r_2(z) \ w_2(y) \ w_2(z) \ r_1(z)$

Is this schedule serializable?

Only two possible serial schedules

$S_1 = r_1(x) \ r_2(y) \ r_1(y) \ r_2(z) \ w_2(y) \ w_2(z)$

$S_2 = r_2(y) \ r_2(z) \ w_2(y) \ w_2(z) \ r_1(x) \ r_1(y) \ r_1(z)$

$S$ is not view equivalent to any serial schedule

Checking view serializability

Detecting view equivalence to a given schedule has linear complexity

Detecting view equivalence to an arbitrary serial schedule is NP complete
- not feasible in real systems

Less accurate but faster techniques should be considered

Conflict equivalence

Conflicting actions
- Action $A_i$ is in conflict with action $A_j$ ($i \neq j$) if both actions operate on the same object and at least one of them is a write
  - Read-Write conflicts (RW or WR)
  - Write-Write conflicts (WW)

Two schedules are conflict equivalent if
- they have the same conflict set
- each conflict pair is in the same order in both schedules
Conflict serializable schedule

- A schedule is **conflict serializable** if it is equivalent to an arbitrary serial schedule of the same transactions.
  - **CSR**: schedules which are conflict serializable

**Example**

\[ S = w_0(x) \ r_1(x) \ w_0(z) \ r_2(z) \ r_3(z) \ w_2(z) \ w_3(x) \]

\[ S_s = w_0(x) \ w_0(z) \ r_3(z) \ r_1(x) \ r_1(z) \ w_2(z) \ r_1(x) \ w_3(z) \ w_2(z) \]

Detecting conflict serializability

- To detect conflict serializability it is possible to exploit the **conflict graph**
- **Conflict graph**
  - a node for each transaction
  - an edge \( T_i \rightarrow T_j \) if
    - there exists at least a conflict between an action \( A_i \) in \( T_i \) and \( A_j \) in \( T_j \)
    - \( A_i \) precedes \( A_j \)
- If the conflict graph is acyclic the schedule is CSR
- Checking graph cyclicity is linear in the size of the graph

**Example of conflict graph**

\[ S = w_0(x) \ r_1(x) \ w_2(x) \ r_3(z) \ r_2(z) \ w_2(z) \ w_3(x) \]

\[ S_s = w_0(x) \ w_1(x) \ r_3(z) \ r_1(x) \ r_1(z) \ w_1(z) \ r_1(x) \ w_2(z) \]

Schedule \( S \) is conflict serializable

Real system settings

- 100 tps (transactions per second)
- each transaction accesses \( \approx 10 \) pages
- each transaction lasts \( \approx 5 \) s

The conflict graph is characterized by 500 nodes

- 100 tps * 5 seconds

Accesses to be checked for conflicts

- 500 nodes * 10 page accessed \( \approx 5000 \) accesses

At each access

- the graph should be updated
- cycle absence should be checked
**VSR versus CRS**

- CSR schedules are a subset of VSR schedules

This schedule is VSR but not CSR

---

**2 Phase Locking**

---

**Locking**

- A lock is a block on a resource which may prevent access to others

**Lock operation**

- Lock
  - Read lock (R-Lock)
  - Write lock (W-Lock)
- Unlock

- Each read operation
  - is preceded by a request of R-Lock
  - is followed by a request of unlock
- Similarly for write operation and W-Lock

---

**Lock manager**

- The scheduler becomes a lock manager
  - It receives transaction requests and grants locks based on locks already granted to other transactions
  - When the lock request is granted
    - The corresponding resource is acquired by the requesting transaction
    - When the transaction performs unlock, the resource becomes again available
  - When the lock is not granted
    - The requesting transaction is put in a waiting state
    - Wait terminates when the resource is unlocked and becomes available

---

**Database Management Systems**
Conflict table

<table>
<thead>
<tr>
<th>Request</th>
<th>Resource State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Free</td>
</tr>
<tr>
<td></td>
<td>R-Locked</td>
</tr>
<tr>
<td></td>
<td>W-Locked</td>
</tr>
<tr>
<td>R-Lock</td>
<td></td>
</tr>
<tr>
<td>W-Lock</td>
<td></td>
</tr>
<tr>
<td>Unlock</td>
<td></td>
</tr>
</tbody>
</table>

Conflict table

<table>
<thead>
<tr>
<th>Request</th>
<th>Resource State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Free</td>
</tr>
<tr>
<td></td>
<td>R-Locked</td>
</tr>
<tr>
<td></td>
<td>W-Locked</td>
</tr>
<tr>
<td>R-Lock</td>
<td>Ok/R-Locked</td>
</tr>
<tr>
<td>W-Lock</td>
<td>Ok/W-Locked</td>
</tr>
<tr>
<td>Unlock</td>
<td>Error</td>
</tr>
</tbody>
</table>

Read locks

- Read locks are shared
  - Other transactions may lock the same resource
  - A counter is used to count the number of transactions currently holding the R-Lock
  - Free when count = 0

Lock manager

- The lock manager exploits
  - the information in the lock table to decide if a given lock can be granted to a transaction
  - stored in main memory
  - for each data object
    - 2 bits to represent the 3 possible object states (free, r_locked, w_locked)
    - a counter to count the number of waiting transactions

2 Phase Locking

- Exploited by most commercial DBMS
- It is characterized by two phases
  - Growing phase
    - needed locks are acquired
  - Shrinking phase
    - all locks are released

CSR
VSR

2 Phase Locking guarantees serializability

A transaction cannot acquire a new lock after having released any lock

2PL

This schedule is not accepted by 2PL but it is serializable
Example

\[ S = r_1(x) \quad w_1(x) \quad r_2(x) \quad w_2(x) \quad r_3(y) \quad w_3(y) \]

- \( T_1 \): Releases the lock on \( x \)
- \( T_1 \): Should acquire a new lock on \( y \)

\( T_1 \) releases the lock on \( x \)
\( T_1 \) should acquire a new lock on \( y \)

\( T_1 \)
\( T_2 \)
\( T_3 \)

\( \text{The schedule is CSR but not 2PL} \)

### Ghost update (a)

<table>
<thead>
<tr>
<th>Transactions</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 ) ( \text{commit} ) ( \text{unlock}_1(x) ) ( \text{unlock}_2(y) ) ( \text{unlock}_2(z) )</td>
<td>( x ) free ( y ) free ( z )</td>
</tr>
<tr>
<td>( T_2 ) ( \text{wait} ) ( w_1(x) ) ( w_1(y) ) ( w_2(z) ) ( w_2(z) ) ( \text{commit} ) ( \text{unlock}_1(y) ) ( \text{unlock}_2(z) )</td>
<td>( 1: ) read ( 2: ) write</td>
</tr>
</tbody>
</table>

\( T_1 \) commits \( x \) \( y \) \( z \)
\( T_2 \) \( \text{wait} \) \( \text{wait} \) \( \text{commit} \) \( \text{unlock}_1(y) \) \( \text{unlock}_2(z) \)

\( T_1 \) releases \( x \) \( y \) \( z \)
\( T_2 \) \( \text{wait} \) \( \text{wait} \) \( \text{commit} \) \( \text{unlock}_1(y) \) \( \text{unlock}_2(z) \)

### Strict 2 Phase Locking

- **Strict 2 Phase Locking allows dropping the commit projection hypothesis**
  - A transaction locks may be released only *at the end* of the transaction
  - After COMMIT/ROLLBACK
- **After the end of the transaction, data is stable**
  - It avoids the dirty read anomaly

### Lock Manager service interface

- **Primitives**
  - \( R\text{-Lock} (T, x, \text{ErrorCode}, \text{TimeOut}) \)
  - \( W\text{-Lock} (T, x, \text{ErrorCode}, \text{TimeOut}) \)
  - \( \text{UnLock} (T, x) \)

- **Parameters**
  - \( T \): Transaction ID of the requesting transaction
  - \( x \): requested resource
  - \( \text{ErrorCode} \): return parameter
    - Ok
    - Not Ok (request not satisfied)
  - \( \text{TimeOut} \): Maximum time for which the transaction is willing to wait

- **Techniques to manage locking**
  - A transaction requests a resource \( x \)
  - If the request *can be satisfied*
    - The lock manager modifies the state of resource \( x \) in its internal tables
    - It returns control to the requesting transaction
  - The processing delay is very small

---

Elena Baralis, Silvia Chiusano
Politecnico di Torino
Techniques to manage locking

- If the request cannot be satisfied immediately
  - The requesting transaction is inserted in a waiting queue and suspended
  - When the resource becomes available
    - the first transaction (process) in the waiting queue is resumed and is granted the lock on the resource
- Probability of a conflict $\approx \frac{K \times M}{N}$
  - $K$ is the number of active transactions
  - $M$ is the average number of objects accessed by a transaction
  - $N$ is the number of objects in the database

When a timeout expires while a transaction is still waiting, the lock manager
- extracts the waiting transaction from the queue
- resumes it
- returns a not ok error code

The requesting transaction may
- perform rollback (and possibly restart)
- request again the same lock after some time
  - without releasing locks on other acquired resources

Hierarchical locking

- Table locks can be acquired at different granularity levels
  - Table
  - Group of tuples (fragment)
    - Physical partitioning criteria
      - e.g., data page
    - Logical partitioning criteria
      - e.g., tuples satisfying a given property
  - Single tuple
  - Single field in a tuple

Hierarchical locking is an extension of traditional locking
- It allows a transaction to request a lock at the appropriate level of the hierarchy
- It is characterized by a larger set of locking primitives
Locking primitives

- Shared Lock (SL)
- eXclusive Lock (XL)
- Intention of Shared Lock (ISL)
  - It shows the intention of shared locking on an object which is in a lower node in the hierarchy
  - i.e., a descendant of the current node
- Intention of eXclusive Lock (IXL)
  - Analogous to ISL, but for exclusive lock

Request protocol

1. Locks are always requested starting from the tree root and going down the tree
2. Locks are released starting from the blocked node of smaller granularity and going up the tree
3. To request a SL or an ISL on a given node, a transaction must own an ISL (or IXL) on its parent node in the tree
4. To request an XL, IXL or SIXL on a given node, a transaction must own an IXL or SIXL on its parent node in the tree

Compatibility matrix

<table>
<thead>
<tr>
<th>Request</th>
<th>ISL</th>
<th>IXL</th>
<th>SL</th>
<th>SIXL</th>
<th>XL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISL</td>
<td>Ok</td>
<td>Ok</td>
<td>Ok</td>
<td>Ok</td>
<td>No</td>
</tr>
<tr>
<td>IXL</td>
<td>Ok</td>
<td>Ok</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SL</td>
<td>Ok</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SIXL</td>
<td>Ok</td>
<td>No</td>
<td>Ok</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>XL</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Precedence graph for locks
Selection of lock granularity

- It depends on the application type
  - if it performs **localized** reads or updates of few objects
    - low levels in the hierarchy (detailed granularity)
  - if it performs **massive** reads or updates
    - high levels in the hierarchy (rough granularity)

Effect of lock granularity

- if it is too coarse, it reduces concurrency
  - high likeliness of conflicts
- if it is too fine, it forces a significant overhead on the lock manager

Predicate locking

- It addresses the ghost update of type b (insert) anomaly
  - for 2PL a read operation is not in conflict with the insert of a new tuple
  - the new tuple can't be locked in advance

- *Predicate locking* allows locking all data satisfying a given predicate
  - implemented in real systems by locking indices

Transaction types

- read-write (default case)
- read only
  - no data or schema modifications are allowed
  - shared locks are enough

The *isolation level* of a transaction specifies how it interacts with the other executing transactions

- it may be set by means of SQL statements

Isolation levels

- SERIALIZABLE
  - the highest isolation level
  - it includes predicate locking
- REPEATABLE READ
  - strict 2PL without predicate locking
  - reads of existing objects can be correctly repeated
  - no protection against ghost update (b) anomaly
    - the computation of aggregate functions cannot be repeated

READ COMMITTED

- not 2PL
- the read lock is released as soon as the object is read
- reading intermediate states of a transaction is avoided
- dirty reads are avoided

READ UNCOMMITTED

- not 2PL
- data is read without acquiring the lock
  - dirty reads are allowed
  - only allowed for read only transactions

Isolation levels

- The isolation level of a transaction may be set by means of the statement
  
  ```
  SET TRANSACTION
  [ISOLATION LEVEL <IsolationLevel>]
  [READ ONLY]
  [READ WRITE]
  ```

- The isolation level may be reduced only for read operations
- Write operations are always executed under strict 2PL with exclusive lock
**Concurrency Control**

**Deadlock**

Typical situation for concurrent systems managed by means of locking and waiting conditions:

- Transaction $T_1$
  - bot
  - $r_{lock}(x)$
  - $r_2(x)$

- Transaction $T_2$
  - bot
  - $r_{lock}(y)$
  - $r_3(y)$

- $w_{lock}(y)$
  - wait

Solving deadlocks:

- **Timeout**
  - The transaction waits for a given time.
  - After the expiration of the timeout, it receives a negative answer and performs rollback.
- Typically adopted in commercial DBMS.
- Length of the timeout interval:
  - Long: long waiting before solving the deadlock.
  - Short: overkill, which overloads the system.

Deadlock prevention:

- **Pessimistic 2PL**
  - All needed locks are acquired before the transaction starts.
  - Not always feasible.
- **Timestamp**
  - Only "younger" (or older) transactions are allowed to wait.
  - It may cause overkill.

Deadlock detection:

- Based on the wait graph:
  - Nodes are transactions.
  - An edge represents a waiting state between two transactions.

- A cycle in the graph represents a deadlock.
- Expensive to build and maintain.
- Used in distributed DBMS.