Concurrency Control

DBMS Architecture

SQL INSTRUCTION

OPTIMIZER

MANAGEMENT OF ACCESS METHODS

BUFFER MANAGER

CONCURRENCY CONTROL

RELIABILITY MANAGEMENT

DATABASE

Index Files
Data Files

System Catalog
Concurrency control

- The workload of operational DBMSs is measured in tps, i.e., transactions per second
  - $\approx 10^{-10^3}$ 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 $x$
    - $r(x)$
  - Write of a single data object $x$
    - $w(x)$
- They may require reading from disk or writing to disk an entire page
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

Transaction $T_1$
- `bot`
- `r_1(x)` $x=2$
- $x=x+1$
- $x=3$

Transaction $T_2$
- `bot`
- `r_2(x)` $x=2$
- $x=x+1$ $x=3$

- `w_2(x)` $x=3$
- `commit`

- `commit`

- `commit`

- $x=4$

- The correct value is $x=4$

- The effect of transaction $T_2$ is lost because both transactions read the same initial value
Dirty read

Transaction T₁
- bot
- $r₁(x)$: $x=2$
- $x=x+1$: $x=3$
- $w₁(x)$: $x=3$

Transaction T₂
- bot
- $r₂(x)$: $x=3$
- $x=x+1$: $x=4$
- $w₂(x)$: $x=4$
- commit

_abort_

cascade rollback

Transaction T₂ reads the value of X in an intermediate state which *never* becomes stable (permanent)

Inconsistent read

Transaction T₁
- bot
- $r₁(x)$: $x=2$
- $r₁(x)$: $x=3$
- commit

Transaction T₂
- bot
- $r₂(x)$: $x=2$
- $x=x+1$: $x=3$
- $w₂(x)$: $x=3$
- commit

Transaction T₁ reads x twice
- x has a different value each time
The correct value is total = 400+200+400=1000

Transaction $T_1$ only partially observes the effect of transaction $T_2$.
**Ghost update (b)**

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

Transaction $T_2$
- bot
- insert a new employee in department $x$ commit

The insert operation is the ghost update

Problem
- The data is *not yet* in the database before the insert
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

Example:

```
T0: r0(x) r0(y) w0(x) r2(x) r2(y) r2(z) r1(y) r1(x) w1(y)
```

T1

T2

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
    - \(r_i(x)\) reads-from \(w_j(x)\) when
      - \(w_j(x)\) precedes \(r_i(x)\) and \(i \neq j\)
      - there is no other \(w_k(x)\) between them
  - final write
    - \(w_i(x)\) 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
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**

\[ S_1 = w_0(x) \cdot r_2(x) \cdot r_1(x) \cdot w_2(x) \cdot w_2(z) \]

\[ S_2 = w_0(x) \cdot r_1(x) \cdot r_2(x) \cdot w_2(x) \cdot w_2(z) \]

- \( S_1 \) is view serializable because it is view equivalent to \( S_2 \)

**View equivalence**

\[ S_3 = w_0(x) \cdot r_2(x) \cdot w_2(x) \cdot r_1(x) \cdot w_2(z) \]

- \( S_3 \) is not view equivalent to \( S_2 \)
  - the reads-from sets are different

\[ S_4 = w_0(x) \cdot r_2(x) \cdot w_2(x) \cdot w_2(z) \cdot r_1(x) \]

- \( S_3 \) is view serializable because it is view equivalent to \( S_4 \)
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
$S = r_1(x) \ r_2(x) \ w_2(x) \ w_1(x)$

Is this schedule serializable?
- Only two possible serial schedules
  - $S_1 = r_1(x) \ w_1(x) \ r_2(x) \ w_2(x)$
  - $S_2 = r_2(x) \ w_2(x) \ r_1(x) \ w_1(x)$

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

Transaction $T_1$
- bot
- $r_1(x)$
- $r_1(x)$
- commit

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

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

---

Inconsistent read anomaly

$S = r_1(x) \; r_2(x) \; w_2(x) \; r_1(x)$

$\triangleright$ Is this schedule serializable?

$\triangleright$ Only two possible serial schedules

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

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

$\triangleright$ S is not view equivalent to any serial schedule
- not serializable
- should be rejected
**Ghost Update (a)**

Transaction $T_1$
- `bot` $r_1(x)$
- $r_1(y)$
- $r_1(z)$
- Total = $x + y + z$
- `commit`

Transaction $T_2$
- `bot` $r_2(y)$
- $y = y - 100$
- $r_2(z)$
- $z = z + 100$
- $w_2(y)$
- $w_2(z)$
- `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*

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

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

*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

\[ S = w_0(x) \ r_1(x) \ w_0(z) \ r_1(z) \ r_2(x) \ r_3(z) \ w_3(z) \ w_1(x) \]

\[ S_s = w_0(x) \ w_0(z) \ r_2(x) \ r_1(x) \ r_1(z) \ w_1(x) \ r_3(z) \ w_3(z) \]

Example

\[ S = w_0(x) \ r_1(x) \ w_0(z) \ r_1(z) \ r_2(x) \ r_3(z) \ w_3(z) \ w_1(x) \]

\[ S_s = w_0(x) \ w_0(z) \ r_2(x) \ r_1(x) \ r_1(z) \ w_1(x) \ r_3(z) \ w_3(z) \]

Schedule \( S \) is conflict serializable.
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

$$S = w_0(x) r_1(x) w_0(z) r_1(z) r_2(x) r_3(z) w_3(z) w_1(x)$$

$T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3$
Example of conflict graph

\[ S = w_0(x) \text{ read}_1(x) w_0(z) \text{ read}_1(z) w_2(z) \text{ read}_3(z) w_3(z) \text{ read}_1(x) \]

\( S \) is CSR (no cycles)

Detecting conflict serializability

- 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
CSR schedules are a subset of VSR schedules.

This schedule is VSR but not CSR.
- **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

**The read lock is** *shared* among different transactions.

**The write lock is** *exclusive*
- it is not compatible with any other lock (R/W) on the same data

**Lock escalation**
- request of R-Lock followed by W-Lock on the same data
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.

The lock manager exploits

- the information in the lock table to decide if a given lock can be granted to a transaction.
- the conflict table to manage lock conflicts.
### Conflict table

<table>
<thead>
<tr>
<th>Request</th>
<th>Resource State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Free</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>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 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

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

Growing phase

Shrinking phase

Locked resources

Time

2 Phase Locking guarantees serializability

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

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

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

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

\( T_1 \)

\( T_2 \)

\( T_3 \)

\( \square \) 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 )</td>
<td>( x ) free ( y ) free ( z ) free ( 1: \text{read} ) ( 1,2: \text{read} ) ( 2: \text{read} ) ( 1,2: \text{read} ) | ( \text{bot} ) ( \text{bot} ) ( 2: \text{read} ) ( \text{wait} ) |</td>
</tr>
<tr>
<td>bot</td>
<td>( r_{\text{lock}}<em>2(y) ) ( r_2(y) ) ( r</em>{\text{lock}}<em>2(z) ) ( r_2(z) ) ( w</em>{\text{lock}}_2(y) ) |</td>
</tr>
<tr>
<td>bot</td>
<td>( r_{\text{lock}}<em>1(x) ) ( r_1(x) ) ( r</em>{\text{lock}}<em>1(y) ) ( r_1(y) ) ( r</em>{\text{lock}}_1(z) ) ( r_1(z) ) |</td>
</tr>
</tbody>
</table>

Elena Baralis, Silvia Chiusano
Politecnico di Torino
**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-Lock (T, x, ErrorCode, TimeOut)
- W-Lock (T, x, ErrorCode, TimeOut)
- UnLock (T, x)

Parameters
- T: Transaction ID of the requesting transaction
- x: requested resource
- ErrorCode: return parameter
  - Ok
  - Not Ok (request not satisfied)
- 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
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 \( \approx (K \times M)/N \)
  - \( K \) 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

Techniques to manage locking

- 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
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

Shared lock and Intention of eXclusive Lock (SIXL)

- Shared lock of the current object and intention of exclusive lock for one or more objects in a descendant node
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>Resource State</th>
<th>ISL</th>
<th>IXL</th>
<th>SL</th>
<th>SIXL</th>
<th>XL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request</td>
<td>ISL</td>
<td>IXL</td>
<td>SL</td>
<td>SIXL</td>
<td>XL</td>
</tr>
<tr>
<td>ISL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IXL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIXL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBMG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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>Ok</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SIXL</td>
<td>Ok</td>
<td>No</td>
<td>No</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

- XL → SIXL
- SIXL → SL, IXL, ISL
- SL, IXL, ISL are mutually incomparable

---

Elena Baralis, Silvia Chiusano
Politecnico di Torino
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
Locking in SQL2 standard

- 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
Isolation levels

- **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

Locking in SQL2 standard

- The isolation level of a transaction may be set by means of the statement:
  
  ```sql
  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
Deadlock

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

Transactions:

Transaction $T_1$
- bot
- $r\_lock_1(x)$
- $r_1(x)$
- $w\_lock_1(y)$

Transaction $T_2$
- bot
- $r\_lock_2(y)$
- $r_2(y)$
- $w\_lock_2(x)$

$\leftarrow wait$
$\leftarrow wait$
Solving deadlocks

Timeout
- the transaction waits for a given time
- after the expiration of the timeout
  - it receives a negative answer and it 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