Transaction Management Part I: Concurrency Control

Shan-Hung Wu & DataLab CS, NTHU

Tx Management

VanillaCore



```
VanillaDb.init("studentdb");
```

```
// Step 1
Transaction tx =
VanillaDb.txMgr().newTransaction(
Connection.TRANSACTION_SERIALIZABLE, true);
```

// Step 2

```
Planner planner = VanillaDb.newPlanner();
String query = "SELECT s-name, d-name FROM •
departments, "
+ "students WHERE major-id = d-id";
Plan plan = planner.createQueryPlan(query,
tx);
Scan scan = plan.open();
```

// Step 3

```
System.out.println("name\tmajor");
System.out.println("-----\t-----");
while (scan.next()) {
  String sName = (String) scan.getVal("s-
  name").asJavaVal();
  String dName = (String) scan.getVal("d-
  name").asJavaVal();
  System.out.println(sName + "\t" + dName);
  }
  scan.close();
```

```
// Step 4
tx.commit();
```

Native API Revisited

- A tx is created upon accepting an JDBC connection
 - by
 VanillaDb.txMgr().newT
 ransaction()
- Passed as a parameter to Planners/Scanners/RecordFiles

Transaction Manager in VanillaDB

• VanillaDb.txMgr() is responsible for creating new transaction and maintaining the active transaction list

TransactionMgr
+ serialConcurMgrCls : Class + rrConcurMgrCls : Class + rcConcurMgrCls : Class + recoveryMgrCls : Class
+ TransactionMgr() + onTxCommit(tx : Transaction) + onTxRollback(tx : Transaction) + onTxEndStatement(tx : Transaction) + createCheckpoint(tx : Transaction) + newTransaction(isolationLevel : int, readOnly : boolean) : Tranasction + newTransaction(isolationLevel : int, readOnly : boolean, txNum : long) : Transaction + getNextTxNum() : long

Transactions

Transaction
+ Iransaction(concurMgr : ConcurrencyMgr,
recoveryMgr : RecoveryMgr,
bufferMgr : BufferMgr
readOnly : boolean, txNum : long)
+ addLifeCycleListener(I : TransactionLifeCycleListener)
+ commit()
+ rollback()
+ endStatement()
+ getTransactionNumber() : long
+ isReadOnly() : boolean
+ concurrencyMgr() : ConcurrencyMgr
+ recoveryMgr() : RecoveryMgr
+ bufferMgr() : BufferMgr

- Ensures ACID
- Concurrency manager for C and I
- Recovery manager for A and D

Transaction Lifecycle

Transaction
+ Transaction(concurMgr : ConcurrencyMgr,
recoveryMgr : RecoveryMgr,
bufferMgr : BufferMgr
readOnly : boolean, txNum : long)
+ addLifeCycleListener(I : TransactionLifeCycleListener)
+ commit()
+ rollback()
+ endStatement()
+ getTransactionNumber() : long
+ isReadOnly() : boolean
+ concurrencyMgr() : ConcurrencyMgr
+ recoveryMgr() : RecoveryMgr
+ bufferMgr() : BufferMgr

- 1. Begin
- 2. End statement
 - If spanning across multiple statements
- 3. Commit or rollback

Lifecycle Listeners

- Tx lifecycle listener
 - Takes actions to tx life cycle events



Lifecycle Listener: Buffer Mgr

- Buffer manager
 - On tx rollback/commit: unpins all pages pinned by the current tx
 - Registered itself as a life cycle listener on start of each tx

```
@Override
public void onTxCommit(Transaction tx) {
    unpinAll(tx);
}
@Override
public void onTxRollback(Transaction tx) {
    unpinAll(tx);
}
@Override
public void onTxEndStatement(Transaction tx) {
        // do nothing
}
```

Lifecycle Listener: Recovery Mgr

- (Naive) Recovery manager
 - Commit: flushes dirty pages and then commit log
 - Rollback: undo all modifications by reading log records

```
@Override
public void onTxCommit(Transaction tx) {
      VanillaDb.bufferMqr().flushAll(txNum);
      long lsn = new CommitRecord(txNum).writeToLog();
      VanillaDb.LogMgr().flush(lsn);
}
@Override
public void onTxRollback(Transaction tx) {
      doRollback(tx);
      VanillaDb.bufferMgr().flushAll(txNum);
      long lsn = new RollbackRecord(txNum).writeToLog();
      VanillaDb.LogMgr().flush(lsn);
}
@Override
public void onTxEndStatement(Transaction tx) {
      // do nothing
}
```

Lifecycle Listener: Concurrency Mgr

- (Naive) Concurrency manager
 - On tx commit/rollback: releases all locks

```
@Override
public void onTxCommit(Transaction tx) {
    lockTbl.releaseAll(txNum, false);
}
@Override
public void onTxRollback(Transaction tx) {
    lockTbl.releaseAll(txNum, false);
}
@Override
public void onTxEndStatement(Transaction tx) {
    // do nothing
}
```

Today's Focus: Concurrency Mgr

VanillaCore

JDBC Interface	JDBC Interface (at Client Side)							
QueryInterface						Jen		
Query Interface							γ	
Тх		Planner Parse						
Storage Interfa	Algebra			gebra				
Concurrency	Recovery	Meta	data	Index	R	ecord	Sql/Util	
		Log Buffer						
		File						

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Outline

• Schedules

- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Concurrency Manager

• Ensures *consistency* and *isolation*

Consistency

Consistency

- Txs will leave the database in a consistent state
- I.e., all integrity constraints (ICs) are meet
 - Primary and foreign key constrains
 - Non-null constrain
 - (Field) type constrain
 - •

- Users are responsible for issuing "valid" txs

Isolation

- Isolation
 - Interleaved execution of txs should have the net effect identical to executing tx in *some* serial order
 - T_1 and T_2 are executed concurrently, isolation gives that the net effect to be equivalent to either
 - T_1 followed by T_2 or
 - T_2 followed by T_1
 - The DBMS does *not* guarantee to result in *which particular* order

Why do we need to interleave txs?

Concurrent Txs

Since I/O is slow, it is better to execute Tx1 and Tx2 concurrently to reduce CPU idle time



- The concurrent result should be the same as serial execution in *some* order
 - Better concurrency

Concurrent Txs

- Pros:
 - Increases throughput (via CPU and I/O pipelining)
 - Shortens response time for short txs
- But operations must be interleaved correctly

- Before executing T_1 and T_2 :
 - A = 300, B = 400

T1: BEGIN A=A+100, B=B-100 END T2: BEGIN A=1.06*A, B=1.06*B END

- Two possible execution results
 - $-T_1$ followed by T_2
 - A = 400, B = 300 → A = 424, B = 318
 - $-T_2$ followed by T_1

• A = 318, B = 424 → A = 418, B = 324

- A schedule is a list of actions/operations from a set of transaction
- If the actions of different transactions are not interleaved, we call this schedule a *serial schedule*

- Equivalent schedules
 - The effect of executing the first schedule is identical to the effect of executing the second schedule
- Serializable schedule
 - A schedule that is equivalent to some serial execution of the transactions

• A possible interleaving schedule

T1:	A=A+100,	B=B-100	
T2:	1	A=1.06*A,	B=1.06*B

- Result: A = 424, B = 318
- A serializable schedule
 - Equivalent to T_1 followed by T_2

• How about this schedule?

T1: A=A+100, B=B-100 T2: A=1.06*A, B=1.06*B

- Result: **A** = **424**, **B** = **324**
- A non-serializable schedule
- Violates the isolation requirement

Goal

- Interleave operations while making sure the schedules are serializable
- How?

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Simplified Notation

T1:	A=A+100,	B=B-100
T2:	A=1.06*A, B=1.06*H	3

• Can be simplified to:

T1:R(A), W(A),R(B), W(B)T2:R(A), W(A), R(B), W(B)

• Here, we care about operations, not values

Anomalies

 Weird situations that would happen when interleaving operations

But not in serial schedules

• Mainly due to the *conflicting* operations

Conflict Operations

 Two operations on the same object are *conflict* if they are operated by different txs and at least one of these operations is a write



Types

- Write-read conflict
- Read-write conflict
- Write-write conflict

T1:
$$R(A)$$
, $W(A)$
T2: $R(A)$, $W(A)$, $R(B)$, $W(B)$

- Read-read conflict?
 - No anomaly

Anomalies due to Write-Read Conflict

- Reading uncommitted data
 - Dirty reads

T1:	R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B)	, W(B)

• A unrecoverable schedule

T1:	R(A), W(A),	R(B), W(B), Abort	
T2:		R(A), W(A), C	

– T1 cannot abort!

- Cascading aborts if T2 completes after T1 aborts

Anomalies due to Read-Write Conflict

• Unrepeatable reads:

$$-T_1: if (A > 0) A = A - 1;$$

-T_2: if (A > 0) A = A - 1;

– IC on A: cannot be negative

T1:	R(A).	R(A), W(A), C
T2:		R(A), W(A), C

T1	A=1				A=0 ,	A=-1,	С	
Т2		A=1,	A=0,	С				

Anomalies due to Write-Write Conflict

• Lost updates:

 $-T_1: A = A + 1; B = B * 10;$

 $-T_2: A = A + 2; B = B * 5;$

– Start with A=10, B=10

T1:	W(A),	W(B), C
T2:	W(A), $W(B)$, C	



Avoiding Anomalies

- Idea:
- Perform all conflicting actions between T1 and T2 *in the same order* (either T1's before T2's or T2's before T1's)
- I.e., to ensure *conflict serializability*

Conflict Equivalent

- If two operations are not conflict, we can *swap* them to generate an equivalent schedule
- Schedule 1 is *conflict equivalent* to schedule 2 and schedule 3



Conflict Serializable

- By swapping non-conflict operations, we can transfer the schedule 1 into a serial schedule 4
- We say that schedule 1 is *conflict serializable*


Ensuring Conflict Serializability is *Not Enough*



• Conflict serializable, but *not* recoverable

Avoiding Anomalies

- We also need to ensure recoverable schedule
- Definition: A schedule is *recoverable* if each tx *T* commits only after all txs whose changes *T* reads, commit
- How?
 - Avoid cascading aborts
 - Disallow a tx from reading uncommitted changes from other txs

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Lock-Based Concurrency Control

- For isolation and consistency, a DBMS should only allow *serializable*, *recoverable* schedules
 - Uncommitted changes cannot be seen (no WR)
 - Ensure repeatable read (no RW)
 - Cannot overwrite uncommitted change (no WW)

A *lock* for each data item seems to be a good solution



$Lock \neq latch$

- Lock: long-term, tx-level
- Latch: short-term, ds/alg-level

Questions

- What type of lock to get for each operation?
- When should a transaction acquire/release lock?

- We need a *locking protocol*
 - A set of rules followed by all transactions for requesting and releasing locks

Two-Phase Locking Protocol (2PL)

- Defines two type of locks:
 - Shared (S) lock
 - Exclusive (X) lock

Compatible?	S	Х
S	True	False
Х	False	False

- Phase 1: Growing Phase
 - Each tx must obtain an S (X) lock on an object before reading (writing) it
- Phase 2: Shrinking Phase
 - A transaction can not request additional locks once it releases any locks
- Ensures conflict serializability

Example



• Ensures conflict serializability

Implementation

- Lock and unlock requests are handled by the lock manager
 - Shared between concurrency managers
- Lock table entry
 - Number of transactions currently holding a lock
 - Type of lock held
 - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations



Lock Table

- Implemented as an inmemory hash table indexed on the name of the data item being locked
- New lock request is added to the end of the queue of requests for the data item
- Request is granted if it is compatible with all earlier requests

From Database System Concepts 6/e, Silberschatz, Korth. Sudarshan.



 Starvation is also possible if concurrency control manager is badly implemented

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

How to improve 2PL to avoid cascading rollback?

Strict Two-Phase Locking

- S2PL
 - Each tx obtains locks as in the growing phase in 2PL
 - 2. But the tx *holds all locks until it completes*
- Allows only serializable and *stric* schedules

Strict Two-Phase Locking

- Definition: A schedule is *strict* iff for any two txs T1 and T2, if a write operation of T1 precedes a conflicting operation of T2 (either read or write), then T1 commits before that conflicting operation of T2
 - Strictiness → no cascading abort (converse not true)
- Avoids cascading rollback, but still has deadlock

Serializability and Recoverability

	view serializable	all schedules
Γ	connect senatizable	recoverable
		avoids cascading abort
	serial	strict

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Coping with Deadlocks

 Cycle of transactions waiting for locks to be released by each other



Other Techniques (1)

- Timeout & rollack (deadlock detection)
 - Assume T_i wants a lock that T_j holds
 - *1.* T_i waits for the lock
 - 2. If T_i stays on the wait list too long then: T_i is rolled back
- *Wait-die* (deadlock prevention)
 - Assume each T_i has a timestamp (e.g., tx number)
 - If T_i wants a lock that T_j holds
 - 1. If T_i is older than T_i , it waits for T_j ;
 - 2. Otherwise T_i aborts

Other Techniques (2)

- **Conservative locking** (deadlock prevention)
 - Every T_i locks **all objects at once** (atomically) in the beginning
 - No interleaving for conflicting txs---performs well only if there is no/very few long txs (e.g., in-memory DBMS)
 - How to know which objects to lock before tx execution?
 - Requires the coder of a stored procedure to specify its read- and write-sets explicitly
 - Does not support ad-hoc queries

You Have Assignment!

Assignment: Conservative Locking

- Implement a ConcurrencyMgr running the conservative locking protocol
 - Modify the stored procedure API to accommodate read-/write-sets

Assignment: Conservative Locking

- Report
 - How you implement the new ConcurrencyMgr
 - API changes and/or new classes
 - Compare the throughputs before and after your modification using the given benchmark & loader
 - Observe and discuss the impact of buffer pool size to your new system

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Granularity of Locks

• What "objects" to lock?

Records vs. blocks vs. tables/files

- Granularity of locking objects
 - Fine granularity: high concurrency, high locking overhead
 - Coarse granularity: low locking overhead, low concurrency

Reducing Locking Overhead

• Data "containers" are nested



 When scanning, can we lock a file instead of all contained blocks/records to reduce the locking overhead?

Multiple-Granularity Locks

- Multiple-granularity locking (MGL) allows users to set locks on objects that contain other objects
 - Locking a file implies locking *all* contained blocks/records
- How does a lock manager know if a file is lockable?
 - Some other tx may hold a conflicting lock on a block in that file

Checking If An Object Is Locked

- To lock a file, check whether all blocks/records in that file are locked
 - Good strategy?
- Does *not* save the locking overhead



Multiple-Granularity Locks

- Allow transactions to lock at each level, but with a special protocol using new "intention" locks:
- Intention-shared (IS)
 - Indicates explicit locking at a lower level of the tree but only with shared locks
- Intention-exclusive (IX)
 - Indicates explicit locking at a lower level with exclusive or shared locks
- Shared and intention-exclusive (SIX)
 - The subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks

Multiple-Granularity Locks

• The compatibility matrix for all lock modes is:

Т

	IS	IX	S	S IX	X
IS	✓	\checkmark	✓	\checkmark	×
IX	✓	\checkmark	×	×	×
S	✓	×	~	×	×
S IX	~	×	×	×	×
Х	×	×	×	×	×

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
 - 1. The lock compatibility matrix must be observed
 - 2. The root of the tree must be locked first, and may be locked in any mode
 - 3. A node *Q* can be locked by *T_i* in S or IS mode only if the parent of *Q* is currently locked by *T_i* in either IX or IS mode
 - 4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode
 - 5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
 - 6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i

Acquiring Locks in MGL: An Example

- Locks are acquired in *root-to-leaf* order
 - Tx1 wants to share-lock a record
 - Tx2 wants to exclusive-lock a file



Releasing Locks in MGL

- Locks need to be released in *leaf-to-root* order
- Why?



Usage Examples of MGL

• T_1 scans R, and updates a few tuples:

- $T_{\rm 1}$ gets an SIX lock on R, and occasionally get X lock on the tuples under modification

- T₂ uses an index to read only part of R:
 - $T_{\rm 2}$ gets an IS lock on R, and repeatedly gets an S lock on a tuple of R
- T_3 reads the size of R:

 $-T_3$ gets an S lock on R

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Dynamic Databases

- So far, we have treated a database as a fixed collection of independent data objects

 Only reads and writes
- However, the database can grow and shrink through the *insertions* and *deletions*
- Any trouble?
- Phantoms
Phantoms Caused by Insertion

- $-T_1$: SELECT * FROM users WHERE age=10;
- $-T_2$: INSERT INTO users

VALUES (3, 'Bob', 10); COMMIT;

 $-T_1$: SELECT * FROM users WHERE age=10;

 A transaction that reads the entire contents of a table multiple times will see different data

– E.g., in a join query

Phantoms Caused by Update

- $-T_1$: SELECT * FROM users WHERE age=10;
- - T_2 : UPDATE users SET age=10 WHERE id=7; COMMIT;
- $-T_1$: SELECT * FROM users WHERE age=10;
- T₁ only share locks the records with the age equals to 10
- The record with id=7 is not in the locking item set of T₁, so T₂ can update this record

How to Prevent Phantoms?

- EOF locks or multi-granularity locks
 - X-lock the containing file when inserting/updating records in a block
 - Hurt performance (no concurrent inserts/updates)
 - Usually used to prevent phantoms by insert
 - But *not* phantoms by update
- Index (or predicate) locking
 - Prevent phantoms caused by both insert and update
 - Works only if indices for the inserting/updating fields are created

Phantom and Conservative Locking

• Phantom problem remains

Assignment bonus: implement MGL to prevent phantom due to inserts

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Transaction Characteristics

- SQL allows users to specify the followings:
- Access model
 - READ ONLY **or** READ WRITE
 - $\mbox{ By Connection.setReadOnly() in JDBC}$

• Isolation level

- Trade anomalies for better tx concurrency
- Ву

Connection.setTransactionIsolation
()

Isolation Levels

• Defined by the ANSI/ISO SQL standard

Isolation level	Dirty reads	Unrepeatable reads	Phantoms
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Read	No	No	Maybe
Serializable	No	No	No

• How to implement these using a locking protocol?

Isolation Levels

• Defined by the ANSI/ISO SQL standard

Isolation level	Dirty reads	Unrepeatable reads	Phantoms
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Read	No	No	Maybe
Serializable	No	No	No

Isolation level	Shared Lock	Predicate Lock
Read Uncommitted	No	No
Read Committed	Released early	No
Repeatable Read	Held to completion	No
Serializable	Held to completion	Held to completion

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Meta-Structures

- DBMS maintains some *meta-structures* in addition to data perceived by users
 - E.g., FileHeaderPage in RecordFile



Concurrency Control of Access to Meta-Structures

- Access to FileHeaderPage?
 - Whenever insertions/deletions of records happen
- How to lock FileHeaderPage?
 S2PL?
- S2PL will serialize all insertions and deletions
 - Hurts performance if we have many inserts/deletes

Early Lock Release

- Actually, lock of FileHeaderPage can be released early
 - No "data" revealed; no hurt to I
- Locking steps for a (logical) insertion/deletion:
 - Acquire locks of FileHeaderPage and target object (RecordPage or a record) in order
 - Perform changes
 - *Release* the lock of FileHeaderPage (but not the object)
- Better concurrency for I
- No harm to C
- Needs special care to ensure A and D

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
 - 2PL and S2PL
 - Deadlock
 - Granularity of locks
- Dynamic databases
 - Phantom
 - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Concurrency Manager

- In storage.tx.concurrency
- Lock-based protocol
 - MGL granularities: file, block, and record
 - S2PL
 - Deadlock detection: time-limit
- Support txs at different isolation levels concurrently
 - Serializable
 - Repeatable Read
 - Read Committed

Lock Modes in Practice (1/2)

 DBMS needs to support concurrent txs in *different* modes

Prevent phantoms due to inserts, but not updates

	Read rec	Modify/delete rec	Insert rec
SERIALIZABLE	IS lock on file and	IX lock on file	X lock on file and
	block	and block	block
	S lock on record	X lock on record	X lock on record
REPEATABLE	IS lock on file and	IX lock on file and	X lock on file and
READ	block; release immediately	block Read committed a	block nd avoid
		cascading abort	
	S lock on record	X lock on record	X lock on record

Lock Modes in Practice (1/2)

	Read rec	Modify/delete rec	Insert rec
READ COMMITTED	IS lock on file and block; release immediately	IX lock on file and block	X lock on file and block
	S lock on record and release it upon end statement	X lock on record Allow non-repeated	X lock on record

Concurrency Manager

• Decide what locks to obtain along the access path

<<abstract>> ConcurrencyMgr

txnum : long
locktbl : Locktable

<<abstract>> + modifyFile(fileName : String) <<abstract>> + readFile(fileName : String) <<abstract>> + insertBlock(blk : BlockId) <<abstract>> + readBlock(blk : BlockId) <<abstract>> + modifyBlock(blk : BlockId)

// methods for B-tree index locking

..

Concurrency Manager

- CCMgr for three isolation levels
 - SerializableConcurrencyMgr
 - -RepeatableRead1ConcurrencyMgr
 - ReadCommittedConcurrencyMgr
- Every transaction has its own concurrency managers corresponding to the isolation level

Lock Table

- Implements the compatibility table
- Use time-limit strategy to resolve deadlock

LockTable
< <final>> ~ IS_LOCK : int <<final>> ~ IX_LOCK : int <<final>> ~ S_LOCK : int <<final>> ~ SIX_LOCK : int <<final>> ~ X_LOCK : int</final></final></final></final></final>
< <synchronized>> ~ sLock(obj: Object, txNum : long) <<synchronized>> ~ xLock(obj: Object, txNum : long) <<synchronized>> ~ sixLock(obj: Object, txNum : long) <<synchronized>> ~ isLock(obj: Object, txNum : long) <<synchronized>> ~ ixLock(obj: Object, txNum : long) <<synchronized>> ~ release(obj: Object, txNum : long, lockType : int) <<synchronized>> ~ releaseAll(txNum : long, sLockOnly : boolean)</synchronized></synchronized></synchronized></synchronized></synchronized></synchronized></synchronized>

References

- Database Design and Implementation, chapter 14. Edward Sciore.
- Database management System 3/e, chapter 16. Ramakrishnan Gehrke.
- Database system concepts 6/e, chapter 15, 16. Silberschatz.
- Derby Developer's Guide: Locking, concurrency, and isolation.
 - http://db.apache.org/derby/docs/10.9/devguide/cdevconcepts 30291.html
- IBM DB2 document: Locks and concurrency control
 - http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.js
 p?topic=%2Fcom.ibm.db2.luw.admin.perf.doc%2Fdoc%2Fc0005
 266.html