

Relational Databases and Datawarehousing Basics of Transaction Management

What do you learn in this chapter ?

- Transactions, Recovery and Concurrency Control
- Transactions and Transaction Management
- Recovery
- Concurrency Control
- The ACID Properties of Transactions

Transactions, Recovery and Concurrency control

- Majority of databases are multi user databases
- Concurrent access to the same data may induce different types of anomalies
- Errors may occur in the DBMS or its environment
- DBMS must support ACID (Atomicity, Consistency, Isolation, Durability) properties

Transactions, Recovery and Concurrency control

- Transaction: set of database operations induced by a single user or application, that should be considered as one undividable unit of work
 - E.g., transfer between two bank accounts of the same customer
- Transaction always 'succeeds' or 'fails' in its entirety
- Transaction renders database from one consistent state into another consistent state

Transactions, Recovery and Concurrency control

- Examples of problems: hard disk failure, application/DBMS crash, division by 0, ...
- **Recovery**: activity of ensuring that, whichever of the problems occurred, the database is returned to a consistent state without any data loss afterwards
- **Concurrency control**: coordination of transactions that execute simultaneously on the same data so that they do not cause inconsistencies in the data because of mutual interference

Transactions and Transaction Management

Delineating transactions and the transaction lifecycle

DBMS components involved in transaction management

Logfile

Delineating Transactions and the Transaction Lifecycle

- Transactions boundaries can be specified implicitly or explicitly
 - Explicitly
 - Developer determines when transaction starts and stops or how steps are rolled back to handle faulty situations
 - **begin transaction** and **rollback transaction** / **commit transaction**
 - Implicitly: first executable SQL statement
- Once the first operation is executed, the transaction is active
- If transaction completed successfully, it can be **committed**. If not, it needs to be **rolled back**.

**HO
GENT**

SP and transactions

```
CREATE PROCEDURE sp_Customer_Insert @customerid varchar(5), @companyname varchar(25), @orderid int OUTPUT
AS
BEGIN TRANSACTION
    INSERT INTO customers(customerid, companyname)
    VALUES(@customerid, @companyname)
    IF @@error <> 0
        BEGIN
            ROLLBACK TRANSACTION
            RETURN -1
        END
    INSERT INTO orders(customerid)
    VALUES(@customerid)
    IF @@error <> 0
        BEGIN
            ROLLBACK TRANSACTION
            RETURN -1
        END
    END
COMMIT TRANSACTION
SELECT @orderid = @@IDENTITY
```

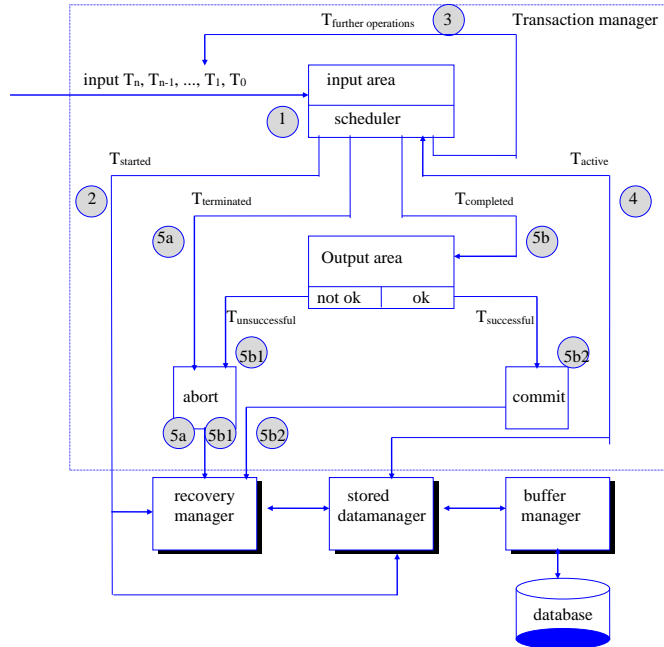

Triggers and transactions

- A trigger is part of the same transaction as the triggering instruction
- Inside the trigger this transaction can be ROLLBACKed
- Although a trigger in SQL Server occurs after the triggering instruction, that instruction can still be undone in the trigger

```
-- If a new record is inserted in OrderDetails => check if the unitPrice is not too low or too high
CREATE OR ALTER TRIGGER insertOrderDetails ON OrderDetails FOR insert
AS
DECLARE @insertedProductID INT = (SELECT ProductID From inserted)
DECLARE @insertedUnitPrice Money = (SELECT UnitPrice From inserted)
DECLARE @unitPriceFromProducts Money = (SELECT UnitPrice FROM Products WHERE ProductID = @insertedProductID)
IF @insertedUnitPrice NOT BETWEEN @unitPriceFromProducts * 0.85 AND @unitPriceFromProducts * 1.15
BEGIN
    ROLLBACK TRANSACTION
    RAISERROR ('The inserted unit price can't be correct', 14,1)
END
```

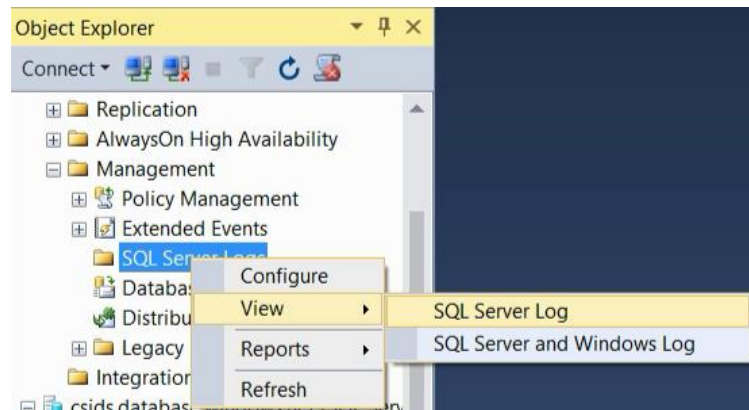
**HO
GENT**

DBMS Components Involved in Transaction Management



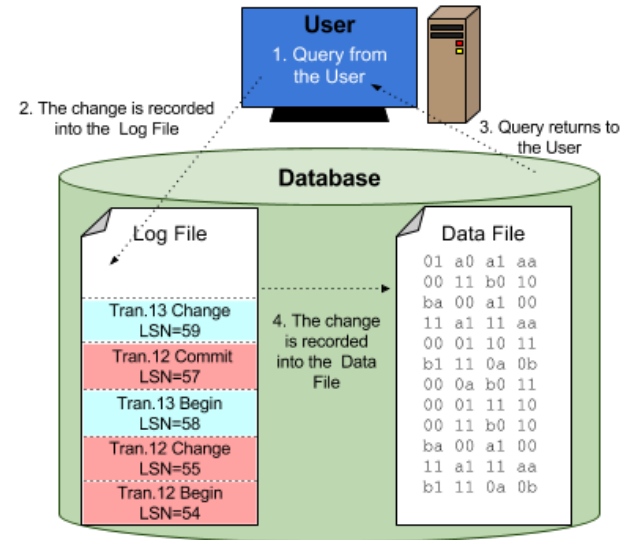
Logfile

- Logfile registers
 - a unique log sequence number
 - a unique transaction identifier
 - a marking to denote the start of a transaction, along with the transaction's start time and indication whether the transaction is read only or read/write
 - identifiers of the database records involved in the transaction, as well as the operation(s) they were subjected to
 - **before images** of all records that participated in the transaction
 - **after images** of all records that were changed by the transaction
 - the current state of the transaction (active, committed or aborted)



Logfile

- Logfile may also contain checkpoints
 - moments when buffered updates by active transactions, as present in the database buffer, are written to disk at once
- Write ahead log strategy
 - all updates are registered on the logfile before written to disk
 - before images are always recorded on the logfile prior to the actual values being overwritten in the physical database files



Logfile

Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

Recovery

Types of Failures

System Recovery

Media Recovery

Types of Failures

- **Transaction failure** results from an error in the logic that drives the transaction's operations and/or in the application logic
- **System failure** occurs if the operating system or the database system crashes
- **Media failure** occurs if the secondary storage is damaged or inaccessible

System Recovery

- In case of system failure, 2 types of transactions
 - already reached the committed state before failure
 - still in an active state
- Logfile is essential to take account of which updates were made by which transactions (and when) and to keep track of before images and after images needed for the UNDO and REDO
- Database buffer flushing strategy has impact on UNDO and REDO

System Recovery

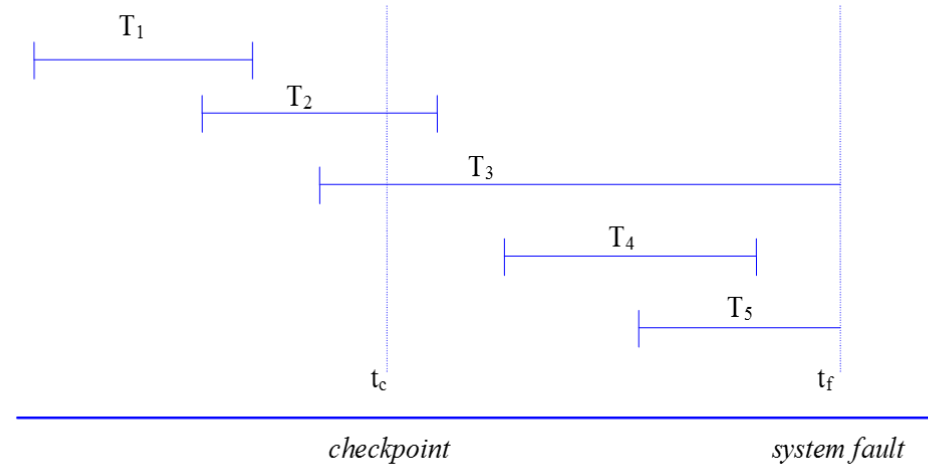
T₁: nothing

T₂: REDO

T₃: UNDO

T₄: REDO

T₅: nothing



Note 1: checkpoint denotes moment the buffer manager last ‘flushed’ the database buffer to disk!

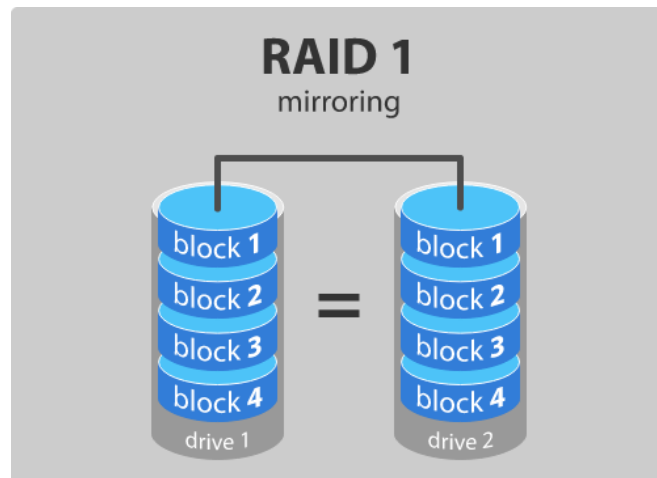
Note 2: similar reasoning can be applied in case of transaction failure (e.g. T₃, T₅)

Media Recovery

- Media recovery is invariably based on some type of data redundancy
 - Stored on offline (e.g., a tape vault) or online media (e.g., online backup hard disk drive)
- Tradeoff between cost to maintain the redundant data and time needed to restore the system
- Two types: disk mirroring and archiving

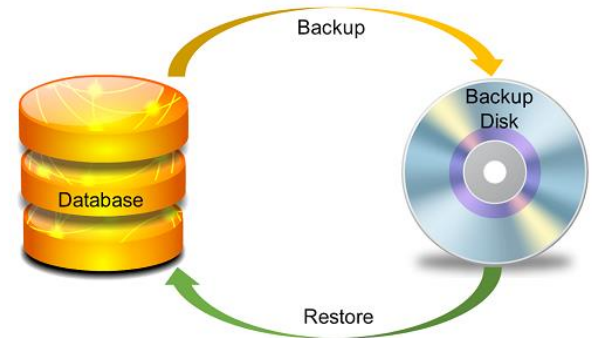
Media Recovery

- Disk mirroring
 - a (near) real time approach that writes the same data simultaneously to 2 or more physical disks
 - limited failover time but often costlier than archiving
 - (limited) negative impact on write performance but opportunities for parallel read access



Media Recovery

- Archiving
 - database files are periodically copied to other storage media (e.g. tape, hard disk)
 - trade-off between cost of more frequent backups and cost of lost data
 - full versus incremental backup



Media Recovery

- Mixed approach: rollforward recovery
 - Archive database files and mirror logfile such that the backup data can be complemented with (a redo of) the more recent transactions as recorded in the logfile
- Note: NoSQL databases allow for temporary inconsistency, in return for increased performance **(eventual consistency)**

Concurrency Control

Typical Concurrency Problems

Schedules and Serial Schedules

Serializable Schedules

Optimistic and Pessimistic Schedulers

Locking and Locking Protocols

Typical Concurrency Problems

- Scheduler is responsible for planning the execution of transactions and their operations
- Simple serial execution would be very inefficient
- Scheduler will ensure that operations of the transactions can be executed in an interleaved way
- Interference problems could occur
 - lost update problem
 - uncommitted dependency problem
 - inconsistent analysis problem

Typical Concurrency Problems

- **Lost update** problem occurs if an otherwise successful update of a data item by a transaction is overwritten by another transaction that wasn't 'aware' of the first update

<i>time</i>	<i>T₁</i>	<i>T₂</i>	<i>amount_x</i>
t ₁		begin transaction	100
t ₂	begin transaction	read(amount _x)	100
t ₃	read(amount _x)	amount _x = amount _x + 120	100
t ₄	amount _x = amount _x - 50	write(amount _x)	220
t ₅	write(amount _x)	commit	50
t ₆	commit		50

Typical Concurrency Problems

- If a transaction reads one or more data items that are being updated by another, as yet uncommitted, transaction, we may run into the uncommitted **dependency** (a.k.a. **dirty read**) problem

<i>time</i>	<i>T₁</i>	<i>T₂</i>	<i>amount_x</i>
t ₁		begin transaction	100
t ₂		read(amount _x)	100
t ₃		amount _x = amount _x + 120	100
t ₄	begin transaction	write(amount _x)	220
t ₅	read(amount _x)		220
t ₆	amount _x = amount _x - 50	rollback	100
t ₇	write(amount _x)		170
t ₈	commit		170

Typical Concurrency Problems

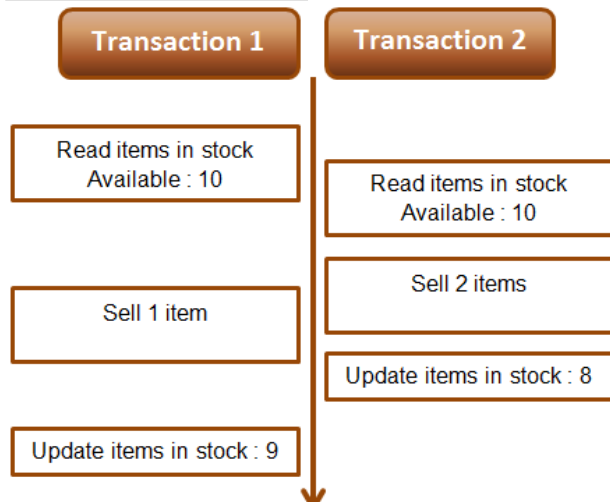
- The **inconsistent analysis** problem denotes a situation where a transaction reads partial results of another transaction that simultaneously interacts with (and updates) the same data items.

time	T_1	T_2	$amount_x$	y	z	sum
t_1		begin transaction	100	75	60	
t_2	begin transaction	sum = 0	100	75	60	0
t_3	read(amount _x)	read(amount _x)	100	75	60	0
t_4	amount _x = amount _x - 50	sum = sum + amount _x	100	75	60	100
t_5	write(amount _x)	read(amount _y)	50	75	60	100
t_6	read(amount _z)	sum = sum + amount _y	50	75	60	175
t_7	amount _z = amount _z + 50	50	75	60	175	
t_8	write(amount _z)	50	75	110	175	
t_9	commit	read(amount _z)	50	75	110	175
t_{10}		sum = sum + amount _z	50	75	110	285
t_{11}		commit	50	75	110	285

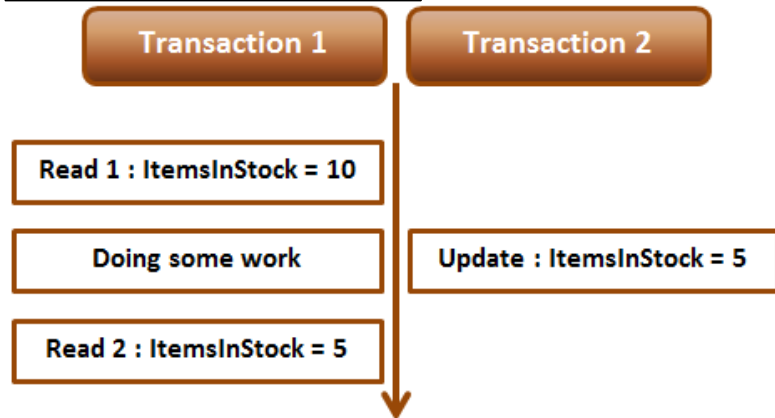
Typical Concurrency Problems

- **nonrepeatable read (unrepeatable read)** occurs when a transaction T_1 reads the same row multiple times, but obtains different subsequent values, because another transaction T_2 updated this row in the meantime
- **phantom reads** can occur when a transaction T_2 is executing insert or delete operations on a set of rows that are being read by a transaction T_1

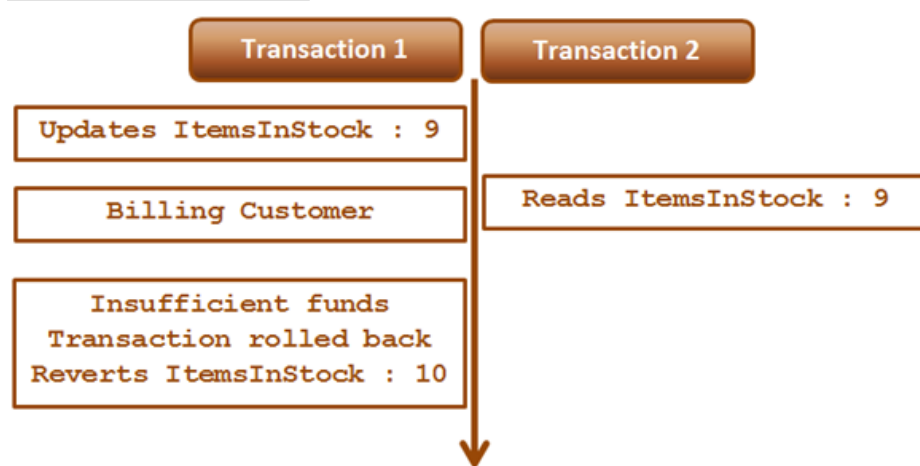
Lost update



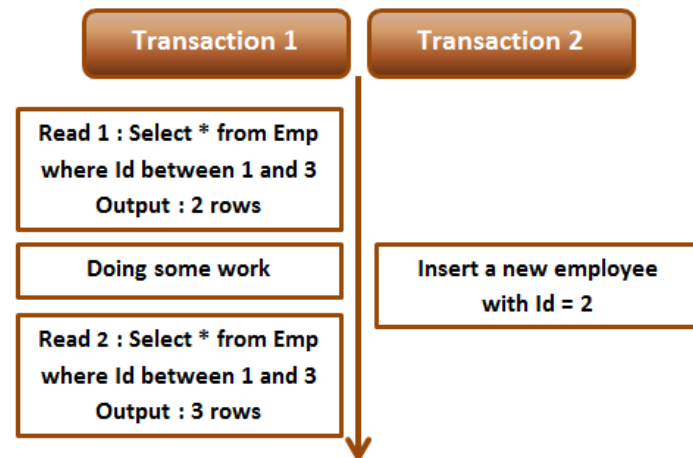
Non repeatable read



Dirty read



Phantom read



Schedules and Serial Schedules

- A *schedule* S is a set of n transactions, and a sequential ordering over the statements of these transactions, for which the following property holds:
“For each transaction T that participates in a schedule S and for all statements s_i and s_j that belong to the same transaction T : if statement s_i precedes statement s_j in T , then s_i is scheduled to be executed before s_j in S .”
- Schedule preserves the ordering of the individual statements *within* each transaction but allows an arbitrary ordering of statements between transactions

Schedules and Serial Schedules

- Schedule S is *serial* if all statements s_i of the same transaction T are scheduled consecutively, without any interleave with statements from a different transaction
- Serial schedules prevent parallel transaction execution
- We need a non-serial, correct schedule!

Schedules and Serial Schedules

- A **serializable** schedule is a non-serial schedule which is equivalent to a serial schedule
- 2 schedules S_1 and S_2 (with the same transactions T_1, T_2, \dots, T_n) are equivalent if
 - *For each operation $read_x$ of T_i in S_1 the following holds: if a value x that is read by this operation, was last written by an operation $write_x$ of a transaction T_j in S_1 , then that same operation $read_x$ of T_i in S_2 should read the value of x , as written by the same operation $write_x$ of T_j in S_2*
 - *For each value x that is affected by a write operation in these schedules, the last write operation $write_x$ in schedule S_1 , as executed as part of transaction T_i , should also be the last write operation on x in schedule S_2 , again as part of transaction T_i .*

Serializable Schedules

	schedule S_1 serial schedule		schedule S_2 non-serial schedule	
time	T_1	T_2	T_1	T_2
t_1	begin transaction		begin transaction	
t_2	read(amount _x)		read(amount _x)	
t_3	amount _x = amount _x + 50		amount _x = amount _x + 50	
t_4	write(amount _x)		write(amount _x)	
t_5	read(amount _y)			begin transaction
t_6	amount _y = amount _y - 50			read(amount _x)
t_7	write(amount _y)			amount _x = amount _x × 2
t_8	end transaction			write(amount _x)
t_9				read(amount _y)
t_{10}		begin transaction		amount _y = amount _y × 2
t_{11}		read(amount _x)		write(amount _y)
t_{12}		amount _x = amount _x × 2	read(amount _y)	end transaction
t_{13}		write(amount _x)	amount _y = amount _y - 50	
t_{14}		read(amount _y)	write(amount _y)	
t_{15}		amount _y = amount _y × 2	end transaction	
t_{16}		write(amount _y)		
t_{17}		end transaction		

Serializable Schedules

- A precedence graph can be drawn to test a schedule for serializability
 - create a node for each transaction T_i
 - create a directed edge $T_i \rightarrow T_j$ if T_j reads a value after it was written by T_i
 - create a directed edge $T_i \rightarrow T_j$ if T_j writes a value after it was read by T_i
 - create a directed edge $T_i \rightarrow T_j$ if T_j writes a value after it was written by T_i

Serializable Schedules

- If precedence graph contains a cycle, the schedule is not serializable.
 - E.g., in the previous example, S_2 contains a cycle

Optimistic and Pessimistic Schedulers

- Scheduler applies scheduling protocol
- Optimistic protocol
 - conflicts between simultaneous transactions are exceptional
 - transaction's operations are scheduled without delay
 - when transaction is ready to commit, it is verified for conflicts
 - if no conflicts, transaction is committed. Otherwise, rolled back.

Optimistic and Pessimistic Schedulers

- Pessimistic protocol
 - it is likely that transactions will interfere and cause conflicts
 - execution of transaction's operations delayed until scheduler can schedule them in such a way that chance of conflicts is minimized
 - will reduce the throughput to some extent
 - E.g., a serial scheduler

Optimistic and Pessimistic Schedulers

- Locking can be used for optimistic and pessimistic scheduling
 - Pessimistic scheduling: locking used to limit the simultaneity of transaction execution
 - Optimistic scheduling: locks used to detect conflicts during transaction execution
- Timestamping
 - Read and write timestamps are attributes associated with a database object
 - Timestamps are used to enforce that a set of transaction operations is executed in the appropriate order

Locking and Locking Protocols

Purposes of Locking

Two-Phase Locking Protocol (2PL)

Cascading Rollbacks

Dealing with Deadlocks

Isolation Levels

Lock Granularity

Purposes of Locking

- Purpose of *locking* is to ensure that, in situations where different concurrent transactions attempt to access the same database object, access is only granted in such a way that no conflicts can occur
- A lock is a variable that is associated with a database object, where the variable's value constrains the types of operations that are allowed to be executed on the object at that time
- Lock manager is responsible for granting locks (*locking*) and releasing locks (*unlocking*) by applying a locking protocol

Purposes of Locking

- An **exclusive lock** (x-lock or write lock) means that a single transaction acquires the sole privilege to interact with that specific database object at that time
 - no other transactions are allowed to read or write it
- A **shared lock** (s-lock or read lock) guarantees that no other transactions will update that same object for as long as the lock is held
 - other transactions may hold a shared lock on that same object as well, however they are only allowed to read it

Purposes of Locking

- If a transaction wants to update an object, an exclusive lock is required
 - only acquired if no other transactions hold any lock on the object
- Compatibility matrix

Type of Lock(s) currently held on object

Type of Lock requested

	<i>unlocked</i>	<i>shared</i>	<i>exclusive</i>
<i>unlock</i>	-	yes	yes
<i>shared</i>	yes	yes	no
<i>exclusive</i>	yes	no	no

Purposes of Locking

- Lock manager implements locking protocol
 - set of rules to determine what locks can be granted in what situation (based on e.g. compatibility matrix)
- Lock manager also uses a lock table
 - which locks are currently held by which transaction, which transactions are waiting to acquire certain locks, etc.
- Lock manager needs to ensure 'fairness' of transaction scheduling to, e.g., avoid starvation

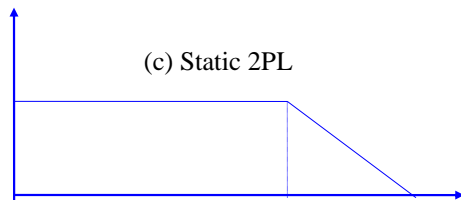
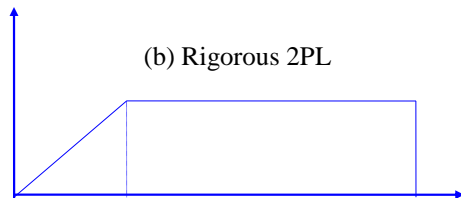
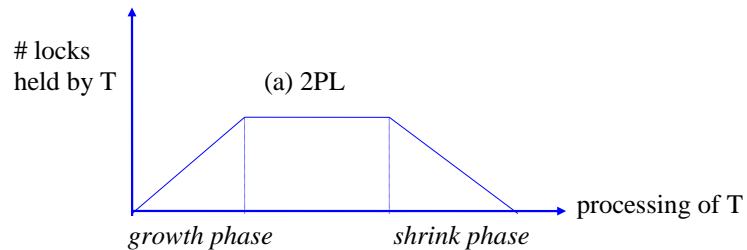
Purposes of Locking

- 2PL locking protocol works as follows:
 1. Before a transaction can read (update) a database object, it should acquire a shared (exclusive) lock on that object
 2. Lock manager determines if requested locks can be granted, based on compatibility matrix
 3. Acquiring and releasing locks occurs in 2 phases
 - growth phase: locks can be acquired but no locks can be released
 - shrink phase: locks are gradually released, and no additional locks can be acquired

Two-Phase Locking Protocol (2PL)

- Variants
 - Rigorous 2PL: transaction holds all its locks until it is committed
 - Static 2PL (Conservative 2PL): transaction acquires all its locks right at the start of the transaction

Two-Phase Locking Protocol (2PL)



Two-Phase Locking Protocol (2PL)

- Lost update problem with locking

<i>time</i>	<i>T₁</i>	<i>T₂</i>	<i>amount_x</i>
t ₁		begin transaction	100
t ₂	begin transaction	x-lock(amount _x)	100
t ₃	x-lock(amount _x)	read(amount _x)	100
t ₄	wait	amount _x = amount _x + 120	100
t ₅	wait	write(amount _x)	220
t ₆	wait	commit	220
t ₇	wait	unlock(amount _x)	220
t ₈	read(amount _x)		220
t ₉	amount _x = amount _x - 50		220
t ₁₀	write(amount _x)		170
t ₁₁	commit		170
t ₁₂	unlock(amount _x)		170

Two-Phase Locking Protocol (2PL)

- Uncommitted dependency problem with locking

<i>time</i>	<i>T₁</i>	<i>T₂</i>	<i>amount_x</i>
t ₁		begin transaction	100
t ₂		x-lock(amount _x)	100
t ₃		read(amount _x)	100
t ₄	begin transaction	amount _x = amount _x + 120	100
t ₅	x-lock(amount _x)	write(amount _x)	220
t ₆	wait	rollback	100
t ₇	wait	unlock(amount _x)	100
t ₈	read(amount _x)		100
t ₉	amount _x = amount _x - 50		100
t ₁₀	write(amount _x)		50
t ₁₁	commit		50
t ₁₂	unlock(amount _x)		50

Cascading Rollback

- Revisit the uncommitted dependency problem
 - problem is resolved if T2 holds all its locks until it is rolled back
 - with 2PL protocol, locks can already be released before the transaction commits or aborts (shrink phase)

<i>time</i>	<i>T₁</i>	<i>T₂</i>	<i>amount_x</i>
t ₁		begin transaction	100
t ₂		x-lock(amount _x)	100
t ₃		read(amount _x)	100
t ₄	begin transaction	amount _x = amount _x + 120	100
t ₅	x-lock(amount _x)	write(amount _x)	220
t ₆	wait	unlock(amount _x)	220
t ₇	read(amount _x)		220
t ₈	amount _x = amount _x - 50	rollback	220
t ₉	write(amount _x)		170
t ₁₀	commit		170
t ₁₁	unlock(amount _x)		170
t ₁₂			

Cascading Rollback

- Before transaction T_1 can be committed, the DBMS should ensure that all transactions that made changes to data items that were subsequently read by T_1 are committed first
- If transaction T_2 is rolled back, all uncommitted transactions T_u that have read values written by T_2 need to be rolled back

Cascading Rollback

- All transactions that have in their turn read values written by the transactions T_u need to be rolled back as well, and so forth
- Cascading rollbacks should be applied recursively
 - can be time-consuming
 - best way to avoid this, is for all transactions to hold their locks until they have reached the 'committed' state (e.g., rigorous 2PL)

Dealing with Deadlocks

- A deadlock occurs if 2 or more transactions are waiting for one another's locks to be released
- Example

<i>time</i>	<i>T</i> ₁	<i>T</i> ₂
t ₁	begin transaction	
t ₂	x-lock(amount _x)	begin transaction
t ₃	read(amount _x)	x-lock(amount _y)
t ₄	amount _x = amount _x - 50	read(amount _y)
t ₅	write(amount _x)	amount _y = amount _y - 30
t ₆	x-lock(amount _y)	write(amount _y)
t ₇	wait	x-lock(amount _x)
t ₈	wait	wait

Dealing with Deadlocks

- Deadlock prevention can be achieved by static 2PL
 - transaction must acquire all its locks upon the start
- Detection and resolution
 - wait for graph consisting of nodes representing active transactions and directed edges $T_i \rightarrow T_j$ for each transaction T_i that is waiting to acquire a lock currently held by transaction T_j
 - deadlock exists if the wait for graph contains a cycle
 - victim selection

Isolation Levels

- Level of transaction isolation offered by 2PL may be too stringent
- Limited amount of interference may be acceptable for better throughput
- Long-term lock is granted and released according to a protocol, and is held for a longer time, until the transaction is committed
- A short-term lock is only held during the time interval needed to complete the associated operation
 - use of short-term locks violates rule 3 of the 2PL protocol
 - can be used to improve throughput!

Isolation Levels


- Reader: statement that reads data using a shared lock (SELECT)
- Writer: statement that writes data, using an exclusive lock (INSERT, UPDATE, DELETE)
- Writers can't be influenced in SQL Server with respect to the locks they claim and the duration of these locks. They always claim an exclusive lock.
 - Readers can be influenced explicitly
 - Using isolation levels

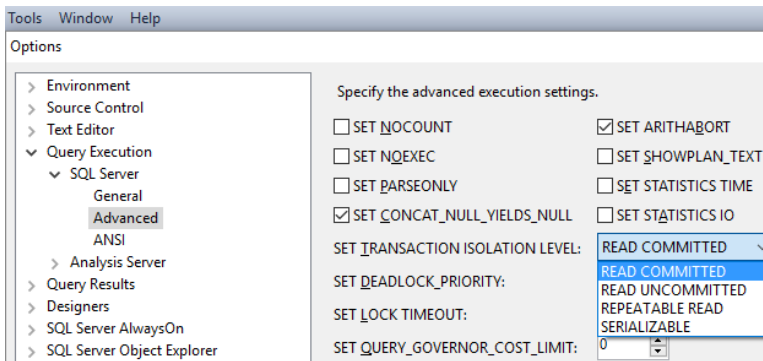
Isolation Levels

- This might have an implicit influence on the behaviour of writers
- Isolation level = setting at session or query level

Isolation Levels in SQL Server

- Based on a pessimistic concurrency control (locking)
 1. READ UNCOMMITTED
 2. READ COMMITTED (default)
 3. REPEATABLE READ
 4. SERIALIZABLE

- 
- locks take longer
 - more consistency
 - less concurrency



Isolation Levels

- **Read uncommitted** is the lowest isolation level. Long-term locks are not taken into account; it is assumed that concurrency conflicts do not occur or simply that their impact on the transactions with this isolation level are not problematic. This isolation level is typically only allowed for read-only transactions, which do not perform updates anyway.
- **Read committed** uses long-term write locks, but short-term read locks. In this way, a transaction is guaranteed not to read any data that are still being updated by a yet uncommitted transaction. This resolves the lost update as well as the uncommitted dependency problem. However, the inconsistent analysis problem may still occur with this isolation level, as well as nonrepeatable reads and phantom reads.

Isolation Levels

- **Repeatable read** uses both long-term read locks and write locks. Thus, a transaction can read the same row repeatedly, without interference from insert, update or delete operations by other transactions. Still, the problem of phantom reads remains unresolved with this isolation level.
- **Serializable** is the strongest isolation level and corresponds roughly to an implementation of 2PL. Now, phantom reads are also avoided. Note that in practice, the definition of serializability in the context of isolation levels merely comes down to the absence of concurrency problems, such as nonrepeatable reads and phantom reads.

Isolation Levels

Isolation level	Lost update	Uncommitted dependency	Inconsistent analysis	Nonrepeatable read	Phantom read
Read uncommitted	Yes	Yes	Yes	Yes	Yes
Read committed	No	No	Yes	Yes	Yes
Repeatable read	No	No	No	No	Yes
Serializable	No	No	No	No	No

Read uncommitted

- Lowest isolation level
- Reader doesn't ask for shared lock
- Reader never in conflict with writer (that holds exclusive lock)
- Reader reads uncommitted data (= dirty read)

Read committed

- Default isolation level
- Lowest level that prevents dirty reads
- Reader reads only committed data
- Reader claims shared lock
- If at that moment a writer holds an exclusive lock, reader has to wait for shared lock

Read committed

- Reader keeps shared lock until data is obtained (end of SELECT), not until end of transaction (= short-term lock)
 - Reading again of data in same transaction can give different result
 - = non-repeatable reads or inconsistent analysis
 - Acceptable for many, but not all applications

Repeatable read

- Reader claims shared lock and holds it until end of transaction
(= long-term lock)
- Other transaction can't get exclusive lock until end of transaction of reader
- Repeatable read = consistent analysis
- Also avoids lost update (possible in 1 & 2) by claiming shared lock at begin transaction (using SELECT because only readers can be influenced, not writers)

Type of Lock(s) currently held on object

	unlocked	shared	exclusive
unlock	-	yes	yes
shared	yes	yes	no
exclusive	yes	no	no

Type of Lock requested

Serializable

- Repeatable read only locks rows found with first SELECT
- Same SELECT in same transaction can give new row (added by other transactions) = phantoms
- Serializable avoids phantoms
- Locks all keys (current and future) that correspond to WHERE-clause

Isolation levels: query level

- Override isolation level with "table hint":

```
SELECT * FROM ORDERS WITH (READUNCOMMITTED);
```

or

```
SELECT * FROM ORDERS WITH (NOLOCK);
```

- This avoids that long running queries on a production system lock updates in other transactions in case of READ COMMITTED and higher

Lock Granularity

- Database object for locking can be a tuple, a column, a table, a tablespace, a disk block, etc.
- Trade-off between locking overhead and transaction throughput
- Many DBMSs provide the option to have the optimal granularity level determined by the database system

ACID Properties of Transactions

- ACID stands for Atomicity, Consistency, Isolation and Durability
- Atomicity guarantees that multiple database operations that alter the database state can be treated as one indivisible unit of work
 - recovery manager can induce rollbacks where necessary, by means of UNDO operations

ACID Properties of Transactions

- Consistency refers to the fact that a transaction, if executed in isolation, renders the database from one consistent state into another consistent state
 - developer is primary responsible
 - also an overarching responsibility of the DBMS's transaction management system

ACID Properties of Transactions

- Isolation denotes that, in situations where multiple transactions are executed concurrently, the outcome should be the same as if every transaction were executed in isolation
 - responsibility of the concurrency control mechanisms of the DBMS, as coordinated by the scheduler

ACID Properties of Transactions

- Durability refers to the fact that the effects of a committed transaction should always be persisted into the database
 - Responsibility of recovery manager (e.g. by REDO operations or data redundancy)

Back-up mechanism

- NEVER use OS backup for a database because
 - Only single or very few database files · incremental backup impossible
 - Data and logfiles are always open
 - Due to running transactions data is inconsistent after restore
 - Use backup tools provided by database vendor: e.g. Microsoft BACKUP command

Back-up mechanism

- On a regular basis the data and logfiles are automatically copied to a safe location
 - Without stopping the system
 - Copies are kept on offline storage
- 2 approaches
 - Complete back-up or Incremental back-up
 - Possible backup strategy: full backup on Sunday night, incremental backup on other nights;
 - Restore: last full backup + subsequent incremental backups, can be very time-consuming => All work of current day is lost!

Back-up

- <https://www.youtube.com/watch?v=l7-6m2cE6JM>

Yesterday,
All those backups seemed a waste of pay
Now my database has gone away

Oh I believe in yesterday

Suddenly,
There's not half the files there used to be
And there's a deadline
hanging over me
The system crashed so suddenly.

I pushed something wrong
What it was I could not say

Now my data's gone
and I long for yesterday-ay-ay-ay.

Yesterday,
The need for back-ups seemed so far away.
Thought all my data was here to stay,
Now I believe in yesterday.

Source

- Principles of Database Management
[Wilfried Lemahieu – Seppe Vanden Broucke – Bart Baesens]