Database Systems (ISYS1001/ISYS5008)

Lecture 9

Part 1: Indexes

Part 2: Transactions

(these are not related topics)

Updated: 6th October,2021

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Copyright Warning

Learning outcomes

- Explain why indexes are important for performance improvement in a database
- Explain the disadvantages related to indexes triggers.
- Implement indexes in a MySQL database
- Explain what is a transaction in a database system and why transactions are important.
- Set transactions with use of COMMIT and ROLLBACK options
- Explain how four different isolation levels are useful in transactions.
- Explain how four different isolation levels are implemented in MySQL

Indexes

Indexes

- *Index* = data structure used to speed access to tuples of a relation, given values of one or more attributes.
- In MySQL indexes are used for :
  - To find the rows matching a WHERE clause quickly.
  - To eliminate rows from consideration. If there is a choice between multiple indexes, MySQL normally uses the index that finds the smallest number of rows
  - To retrieve rows from other tables when performing joins.
  - To find the MIN() or MAX() value for a specific indexed column
  - ….
- Could be a hash table, but in a DBMS it is a balanced search tree with giant nodes (a full disk page) called a *B-tree*.

Declaring Indexes

- No standard in SQL

- MySQL syntax:

```
CREATE INDEX <name>
  ON <Table>(<attribute list>);
```

Using Indexes in MySQL

- Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index.

- Example: use *HardInd* and *BookInd* to find the isbn numbers and titles of all hardcover books that have a dust jacket that include the descriptive word ‘Dragon’ and whose titles include the word ‘Manual’.

```
CREATE INDEX HardInd ON Hardcover(jacket);
CREATE INDEX BookInd ON Books(title);
```

Using Indexes

```
SELECT isbn, title
FROM Books NATURAL JOIN Hardcover
WHERE title LIKE '%Manual%'
AND jacket LIKE '%Dragon%';
```

1. Use *BookInd* to get all books with a title including the word ‘Manual’.
2. Then use *HardInd* to get those hardcovers that have a jacket description including the word ‘Dragon’.

Database Tuning

- A major problem in making a database run fast is deciding which indexes to create.
- Pro: An index speeds up queries that can use it.
- Con: An index slows down all modifications on its relation because the index must be modified too.
- Note: An index speeds up UPDATE statements if the indexed attribute is in the WHERE column but will slow them down in general.

Example: Tuning

- Suppose the only things we did with our books database was:
  1. Insert new facts into books and either Softcovers or Hardcovers (20%).
  2. Look up softcover books (70%).
  3. Look up hardcover books (10%).

- Then an index on Softcover would be wonderful, but an index on Hardcovers would be harmful.

Tuning Advisors

- A major research thrust.
  - Because hand tuning is so hard.
- An advisor gets a *query load*, *e.g.*:
  1. Choose random queries from the history of queries run on the database, or
  2. Designer provides a sample workload.
- The advisor generates candidate indexes and evaluates each on the workload.
  - Feed each sample query to the query optimizer, which assumes only this one index is available.
  - Measure the improvement/degradation in the average running time of the queries.

Transactions

Concurrent access issues

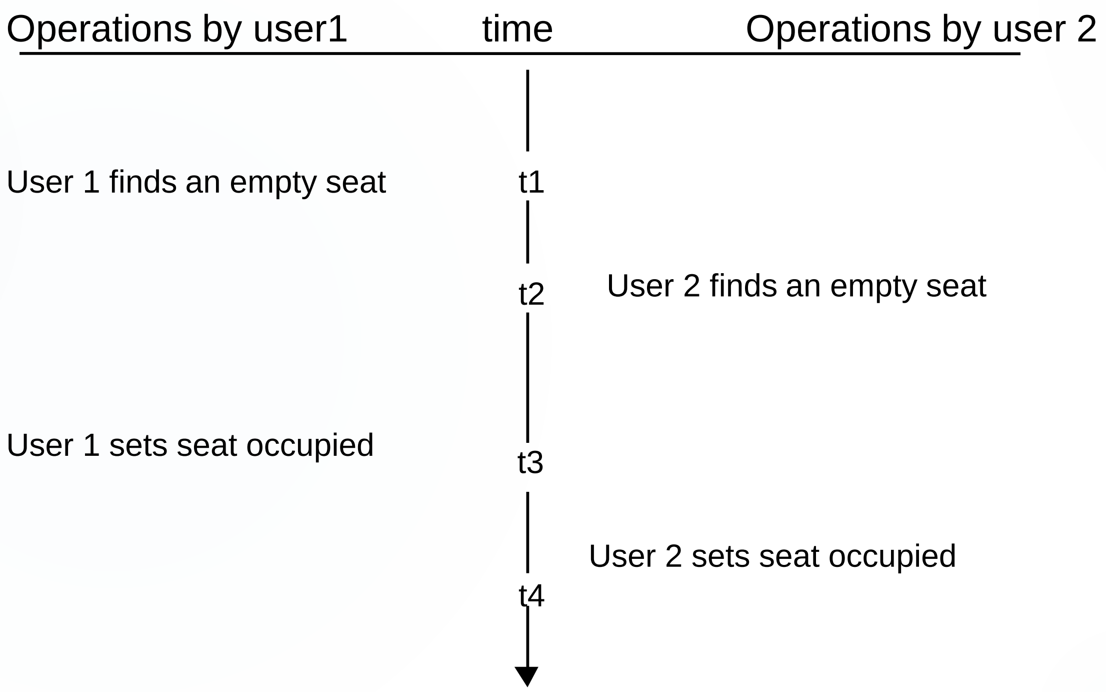
- In our practical classes we have a set up of individual user and each user run a single query at a time.
- Database status left by one operation is the state upon which the next operation acts.
- In real world scenarios, database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.
  - Hundreds of operations per second
- Examples: airline reservations, banking, hotel booking
  - Operations initiated from many thousands of sites (ATMs, travel agents, airline or bank employees, etc.)
  - Two operations can affect the same flight or account and these operations can overlap in time
- However, while many concurrent access happens, database should make sure that it is not in a status which cannot be explained as result of the operations performed on it.

Example: Interactions

- Assume two people are having a common account and each take $100 out of it, from different ATM's at about the same time.
  - The DBMS better make sure one account deduction doesn't get lost.
- Compare:
  - An OS allows two people to edit a document at the same time.
  - If both write, the changes of one get lost.

Lost update problem

- Consider two users trying to book the same seat on a flight.
  - Suppose the users simultaneously execute a function from different locations, to choose an available seat.
  - The function accesses the same information in the DB.
  - The following scenario could occur.



Operations by user1	time	Operations by user 2
User 1 finds an empty seat	t1	
	t2	User 2 finds an empty seat
User 1 sets seat occupied	t3	
	t4	User 2 sets seat occupied

Lost update problem

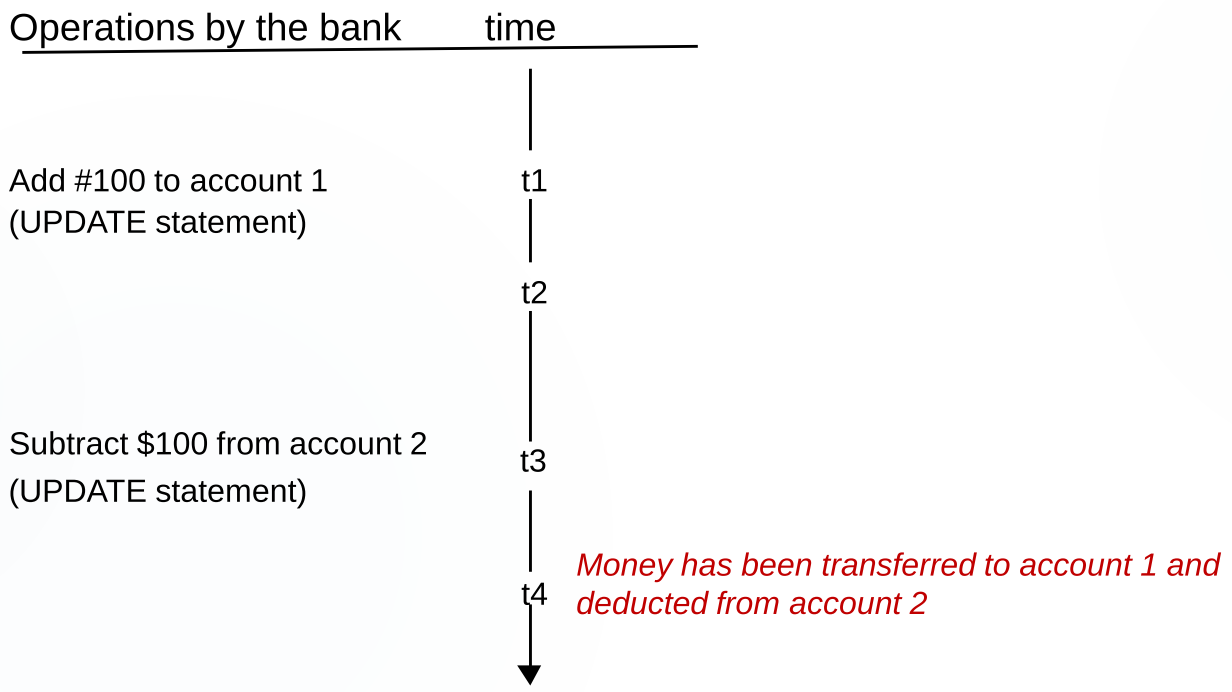
- Each user could perform the operations correctly, but the global result is incorrect.
  - Both customers think they have been given the requested seat.
  - Problems related to non-serialized behavior
- Controlling the sequence in which different operation which affect the database occur helps to reduce issues related to concurrent events.
- Solution?
  - Serialize the two function executions based on the concept of transactions.
  - Transaction: group of operations that have to be performed together. (more later)

Serial and serializable

- Serial: One transaction executes completely before any other transaction begins.
  - If the transactions were run serially, the error of lost update could not occur.
  - In practice, need some parallelism and so cannot run all operations serially.
- Serializable:
  - Operations may be interleaved, but execution must be equivalent to *some* sequential (serial) order of all transactions

Atomicity related problems

- We want to transfer $100 from account 1 to account 2.
- First , we have to check whether there is $100 available in the account 1 . If so , following operations can occur:



Operations by the bank time

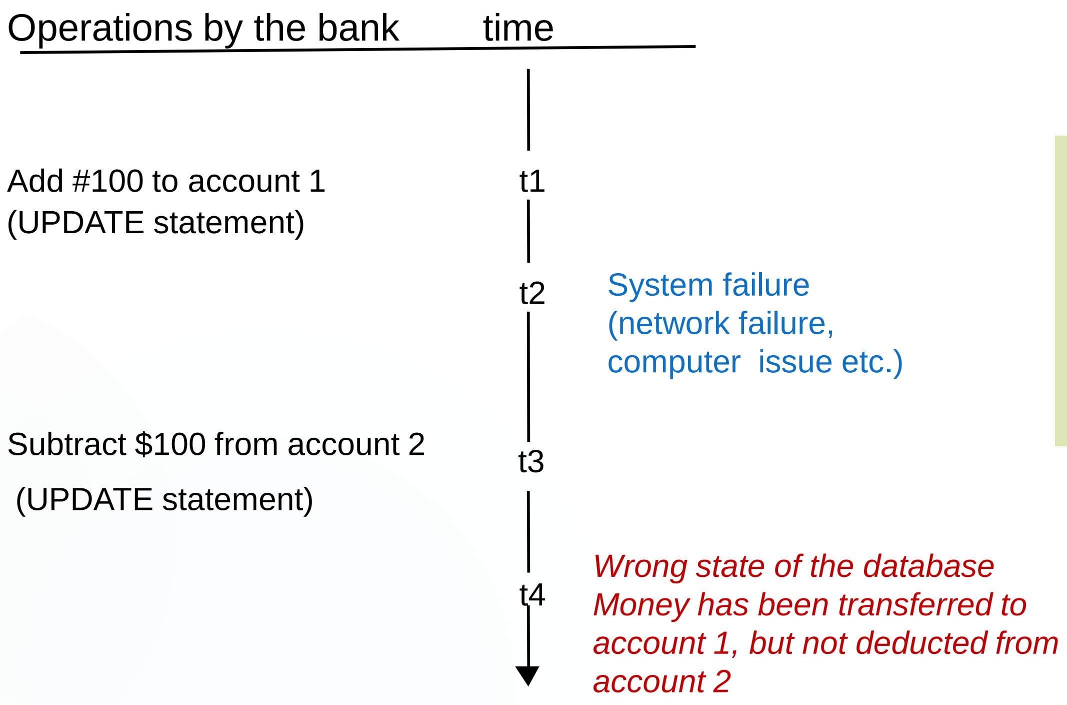
Add #100 to account 1
(UPDATE statement)

Subtract $100 from account 2
(UPDATE statement)

Money has been transferred to account 1 and deducted from account 2

Atomicity related issues

- If a system failure occur after the first UPDATE statement and before the second UPDATE statement, final state of the database will be incorrect.



Add #100 to account 1 (UPDATE statement)

System failure (network failure, computer issue etc.)

Subtract $100 from account 2

(UPDATE statement)

Wrong state of the database Money has been transferred to account 1, but not deducted from account 2

*Certain combinations of database operations needs to be done **atomically**, i.e. , **either they are both done or neither is done**.*

- Either the second account should be credited with the amount or the change to the first account should be reversed. The two updates are to be done atomically – either both are done or none are.
- *Regarding the previous example of flight booking also, while seat selection is happening, system failures may occur*

Atomicity : possible solution

- All the changes are done in a local workspace and only after all work is done, *commit* the work in the database.

Transactions

- Problems of non-serialization and non-atomicity can be solved by grouping database operations into transactions.
- A transaction is a collection of one or more operations on the database that must be executed *atomically* (either all or none) . Further, SQL requires by default, all transactions are executed in a *serializable* manner.
- Transaction*(Xact)* is a basic concurrency/recovery concept*.*
- A transaction is a sequence of one or more SQL operations treated as a unit.
  - Transactions appear to run in isolation
  - If the system fails, each transaction's changes are reflected either entirely or not at all

ACID properties of Transactions

- *Atomic* : Each transaction is “all-or-nothing,” never left half done.
- *Consistent* : All constraints hold when each transaction begins, must guarantee all constraints hold when transaction ends
- *Isolated* : It appears to the user as if only one transaction executes at a time.
- *Durable* : If system crashes after transaction commits, all effects of transaction remain in database.

Transactions in SQL

- In a SQL interface, each statement is a transaction by default(“autocommit”).
- However, SQL allows grouping several statements into a single transaction also.
- START TRANSACTION is used to mark the beginning of a transaction.
- COMMIT ends a transaction successfully;

  Any changes done to the database since the current transaction began are permanently written to the database; Before COMMIT is executed, changes are tentative and may or may not be visible to other transactions.
- ROLLBACK causes the transactions to abort, or terminated unsuccessfully.

  Any changes done after starting the transaction are undone.
  - Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.
- Current transaction ends on session termination.

- *By default, MySQL runs with autocommit mode enabled. This means that, when not otherwise inside a transaction, each statement is atomic, as if it were surrounded by START TRANSACTION and COMMIT.*

Achieving serializable execution

- Common approach is to lock database elements.
  - Prevents two functions accessing the same element for update at the same time.
  - If the function to choose a seat locked the flights relation, operations that did not access flights could run in parallel.
- Variations of locks: Shared and exclusive locks
- Other methods are also useful (e.g., time stamping)

Lock types

- Lock info maintained by a "lock manager":
  - Stores (TransID, RecID, Mode) in a lock table.
  - Mode ∈ {S,X}
  - Lock compatibility table:
- If a transaction can't get a lock, it is suspended on a wait queue.

	--	S	X
--	√	√	√
S	√	√	
X	√		

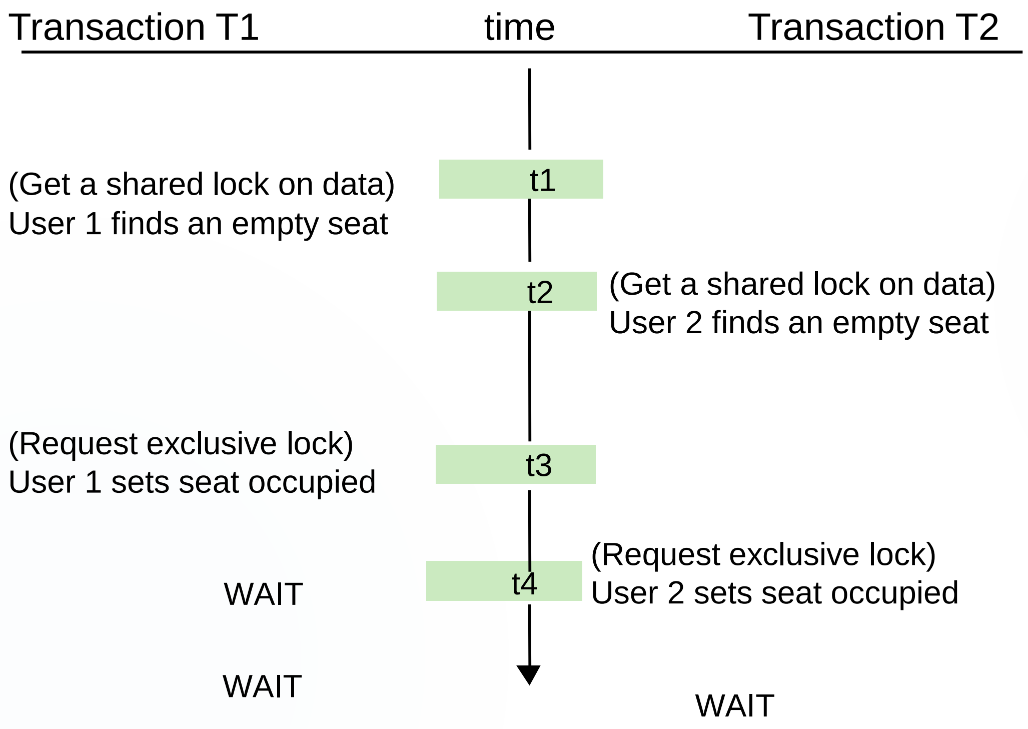
The uncommitted dependency problem

Transaction T1	time	Transaction T2
	t1	UPDATE p
RETRIEVE p	t2	
	t3	ROLLBACK
	t4	

solution with Locking

Transaction T1	time	Transaction T2
	t1	UPDATE p (X lock on p)
RETRIEVE p (request S lock on p)	t2	
WAIT	t3	ROLLBACK (release X lock on p)
resume: RETRIEVE p (S lock on p)	t4	

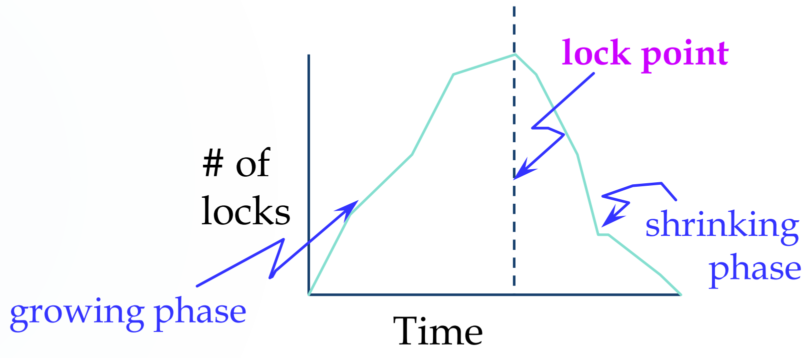
Some problems with Locks



Transaction T1	time	Transaction T2
(Get a shared lock on data) User 1 finds an empty seat	t1	
	t2	(Get a shared lock on data) User 2 finds an empty seat
(Request exclusive lock) User 1 sets seat occupied	t3	
WAIT	t4	(Request exclusive lock) User 2 sets seat occupied
WAIT		WAIT

Two-Phase Locking (2PL)

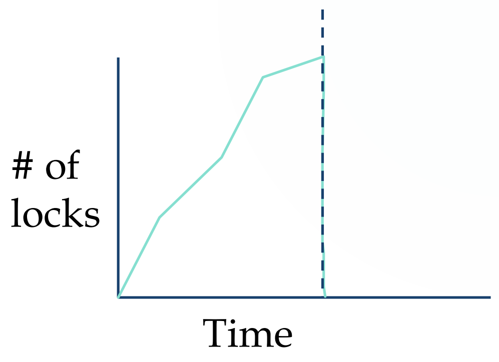
- 2PL:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - If T releases any lock, it can acquire no new locks!
- Locks are automatically obtained by DBMS.
- *Guarantees* serializability!
  - Why?



Strict 2PL

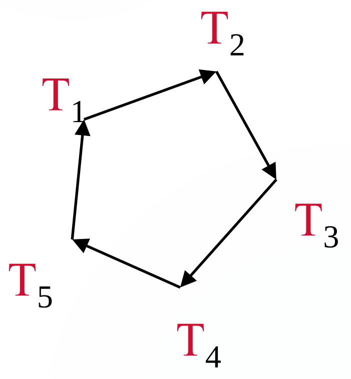
- Strict 2PL:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - Hold all locks until end of transaction.
- Guarantees serializability
  - Also, abort of a transaction does not require cascade abort of other transactions
- Avoids uncommitted dependency problem

  (no overwriting of uncommitted data)

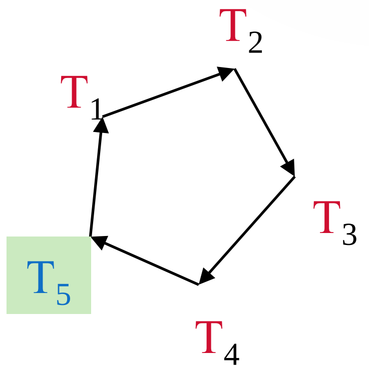


Deadlock Detection and resolving

- Lock Mgr maintains a "Waits-for" graph:
  - Node for each Xact.
  - Arc from T_i to T_j if T_j holds a lock and T_i is waiting for it.
- Periodically check graph for cycles.



- Resolving Deadlocks:
  - "Shoot" some Xact to break the cycle.
    - T5 is chosen as "victim"
    - T4 can now proceed
    - T5 will be restarted
  - Simpler hack: *time-outs*.



With 2P Locking

Transaction T1	time	Transaction T2
(Get a S lock on data) User 1 finds an empty seat	t1	
	t2	(Request a S lock on data) User 2 wants to finds an empty seat
		WAIT
User 1 sets seat occupied (Release X lock) (Release S lock)	t3	
	t4	(User 2 gets S lock) User 2 finds empty seats; seat occupied by User 1 not appears
	t5	(Release S lock) User 2 aborts the work without reserving a seat

Dirty Reads

- Dirty data is a term used for data written by a transaction that has not yet committed.
- A dirty read is a read of dirty data.
- Risk of dirty read:
  - The transaction that wrote the data may abort.
  - Then the database will remove the dirty data as if it never existed.
  - A transaction that read the dirty data may perform incorrect computation because of it.

Dirty Reads

- Variation of seat choosing example:
  1. Step1 : We find an empty seat (‘s’), then reserve it by setting the seat status of that seat to “occupied”. If no available seat, end the work.
  2. Step2 : We ask the user (user 1) to confirm the seat by approving it. If approved, then the transaction is ***committed***. If not seat is released by setting the seat status to “available’ and repeat step1 to get another seat .
- With dirty reads : Let two users are doing the same steps above at about the same time. User 1 may reserve seat ‘s’, which may be later rejected(not confirmed). If a second user performs step1 above, before the rejection by the first user, seat ‘*s*’ may not be visible to the second user. (dirty read occurs)
- In this case, *dirty reads cause less issues* as, even if the user 2 cannot see the seat ‘*s*’ which is available but still, if we allow the user 2 to query again, seat ‘*s*’ can be seen.
- However, in our bank transaction example, dirty reads is not be a good choice.

Allowing dirty data (isolation levels)

- SQL allows to specify that dirty reads are allowed.

```
SET TRANSACTION READ WRITE

   ISOLATION LEVEL READ UNCOMMITED;
```

  - Line 1 : Transaction may write data (default is read for this isolation level)
  - Line 2 : Transaction may run with the "isolation level" read-uncommitted. That is, the transaction is allowed read dirty data.

- Xact T2 can see dirty data from T if T2 is running at level READ UNCOMMITTED and T aborts.

Isolation levels for transactions

- SQL allows four isolation levels
- Isolation levels can be different for each transaction
- Possible levels:
  1. SERIALIZABLE
  2. READ UNCOMMITTED
  3. READ COMMITTED
  4. REPEATABLE READ

As we move down the list:

↓ Overhead, ↓ Consistency Guarantees

↑ Concurrency

Isolation levels for transactions

- Isolation level of a Xact affects only what data that Xact may see
  - Does not affect what any other Xact sees
  - If T is running at level SERIALIZABLE, the execution of T appears as if all other Xacts run entirely before or entirely after T.
  - If some Xacts are running at another isolation level, they may see the data written by T as T writes it.
  - A Xact T2 can even see dirty data from T if T2 is running at level READ UNCOMMITTED and T aborts.

Transactions at different isolation levels

- READ-COMMITED isolation level:

```
SET TRANSACTION READ ONLY

   ISOLATION LEVEL READ COMMITTED
```

  - Does not allow to have dirty reads (not allowed to read uncommitted data)
  - However, it allows a transaction runs at this isolation level to issue the same query many times and get different answers, as long as the answers reflect the data written by the transactions already committed.
  - Default is read-write for READ COMMITTED, so if we need we have to set it to READ ONLY.
  - Does not guarantee global serializability.
- Seat choosing example with READ COMMITTED:
  - User 2 may not see seat ‘s’ as booked , if the user 1 (first user) reserve the seat ‘s’ but not committed.

Transactions at different isolation levels

- REPEATABLE-READ

```
SET TRANSACTION READ ONLY

   ISOLATION LEVEL REPEATABLE READ
```

  - A transaction may NOT perform dirty reads.
  - Default is read-write for READ COMMITTED, so if we need we have to set it to READ ONLY.
  - An item read multiple times and assume that an identical tuple will be retrieved.
  - However, a relation can change (new inserts) while the transactions are going on. – *phantom tuples*

    (same query produces different sets of rows at different times)
  - Does not guarantee global serializability
- Seat choosing example with REPEATABLE-READ:
  - User 2 may not see seat ‘s’ as booked, if the user 1 (first user) reserve the seat ‘s’ but not committed. In subsequent repeating queries also seat ‘s’ will not be available,
  - If some new tuples are entered to the relation (assume the actual plane is changed and more seats make available) , the new tuples will be retrieved by repeating query for the seats.

Setting Transactions in MySQL

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [, transaction_characteristic]
   ...
transaction_characteristic: {
    ISOLATION LEVEL level
  | access_mode
}
level: {
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
}
access_mode: {
     READ WRITE
   | READ ONLY
}
```

Example: Isolation Level Read Uncommitted

- A transaction may perform dirty reads
- Consider two concurrent transactions:

T1:
```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
   UPDATE Emp
    SET salary = salary * 1.1
    WHERE job = ‘Software Designer’;
```

T2:
```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
   SELECT AVG(salary)
    FROM Emp;
```

Example: Isolation Level Read Committed

- A transaction may NOT perform dirty reads
  - Does not guarantee global serializability
- Consider two concurrent transactions:

T1:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
   UPDATE Emp
    SET salary = salary * 1.1
    WHERE job = ‘Software Designer’;
```

T2:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
   SELECT AVG(salary) FROM Emp; -- before T1
   SELECT MAX(salary) FROM Emp; -- after T1
```

Example: Isolation Level Repeatable Read

- A transaction may NOT perform dirty reads
- Still does not guarantee global serializability
- Consider two concurrent transactions:

T1:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
   UPDATE Emp SET salary = salary * 1.1;
   UPDATE Emp SET job = ‘Software Designer’
    WHERE empno = ‘12345’;
```

T2:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
   SELECT AVG(salary) FROM Emp;  -- before T1
   SELECT COUNT(*) FROM Emp
    WHERE job = ‘Software Designer’; -- after T1
```

Isolation levels: summary

	dirty reads	non-repeatable reads	phantoms
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable Read	N	N	Y
Serializable	N	N	N

Summary

- Transactions are important to manage concurrent access and issues related to non-atomicity of database operations.
- Transactions are having ACID properties.
- Serial execution is the model of correctness when executing transactions; Serializability allows us to “simulate” serial execution with better performance.
- Locks are useful to get seralizability.
  - 2PL: A simple mechanism to get serializability.
  - Strict 2PL also gives us recoverability.
- Locks can leads to deadlocks, and typically a deadlock detector is used to solve the problem.
- Dirty reads are allowed to use uncommitted data of a trnsaction.
- SQL allows three other weaker isolation levels than SERIALIZABLE. They are, READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ

References

- Indexes MySQL reference:

https://dev.mysql.com/doc/refman/8.0/en/create-index.html

- Additional information on Locking:

https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html

- Additional information on Phantom tuples:

https://dev.mysql.com/doc/refman/8.0/en/innodb-next-key-locking.html

Happy Database systems

Next week : Working with a programming language

Practical worksheet 10 (a short practical sheet)

Prac test 4 will be held in the next week