Part 16: Concurrency Control

Elmasri/Navathe:

- Chap. 19, "Transaction Processing Concepts"
- Chap. 20, "Concurrency Control Techniques"

I have also used the following sources:

- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Edition, McGraw-Hill, 1999. Chap. 13, "Transactions", and Chap. 14, "Concurrency Control".
- Kemper/Eickler: Datenbanksysteme (in German), Chap. 11, Oldenbourg, 1997.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Gray/Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, 1993.
- Oracle8 Concepts, Release 8.0, Oracle Corporation, 1997, Part No. A58227-01.
- Oracle8 Application Developer's Guide, Oracle Corporation, 1997, Part No. A58241-01.

Objectives

- You should understand the goal of isolation/serializability.
- You should understand why interleaved execution of transactions improves the performance.
- Should should understand the notions of locks (read/write).
- Should should be able to explain some classes of problems due to interleaved execution, and you should know whether the DBMS automatically does the necessary locking or whether you have to use "`FOR UPDATE`" etc.
- You should know what a deadlock is.

Overview

Goal: Isolation

- Every user should have the impression that he/she has exclusive access to the database for the duration of the entire transaction.
- I.e. all other transactions must appear to him/her as either completely before his/her transaction, or completely after it.
- So what users see and the changes they perform must be equivalent to a serial schedule of transactions (as if there were only one terminal from which the DB can be accessed).
- If the DBMS guarantees isolation, application programming is simplified, because the programmer does not have to worry about possible other transactions running concurrently.

Goal: Performance

- While one transaction waits for the disk or user input, the DBMS should work on another transaction.
- A long running transaction must be interrupted from time to time to give shorter transactions a chance. This gives overall shorter response times: Otherwise many short transactions will queue up after one long one.
- When a transaction commits, log entries must be written to disk. If we allow transactions to run concurrently, and delay the commit a bit, writing a single block to the log might be sufficient for the commit of multiple transactions.
- Concurrent transactions might make use of parallel hardware.

Problems

- The two goals are in conflict with each other:
  If we really want 100% isolation, we will end up with very little parallelism — often we have to lock entire tables.
- SQL has no "begin transaction" command. As long as there are only queries, it is not clear whether they form one big transaction or are each a transaction in itself.
- DBMS guarantee "some isolation" and offer the mechanisms to reach full isolation, but need help from the pogrammer.
- Errors due to concurrent execution will not be noticed during normal debugging — real system load is needed and even then it may take months until the fatal situation occurs.

Locks (1)

- Most systems use locks for concurrency control.
- Locks can be used for objects of different granularity: Tables, blocks, tuples, attributes.
- If Transaction A holds a lock on an object, and another Transaction B also wants to lock this object, B must wait.

  B is suspended from execution (put to sleep). The lock manager has for each lock a list of waiting transactions. So when Transaction A takes the lock off, the lock manager can wake B up. Then B can acquire the lock.
- You can try this by logging into SQL*Plus from different windows and entering the commands on the next page.

Locks (2)

Transaction A	Transaction B

`UPDATE Emp`
`SET Sal = Sal * 2`
`WHERE EName = 'JAMES'`
⟶ 1 row updated.	

	`UPDATE Emp`
	`SET Sal = Sal / 2`
	`WHERE EName = 'JAMES'`
	⟶ (no reaction)

`COMMIT`	
	⟶ 1 row updated.

Locks (3)

- Often DBMS have (at least) two kinds of locks:
- Exclusive locks (X) are used for write accesses, and exclude any other access (read or write).
- Shared locks (S) are used for read accesses. They exclude write accesses, but allow read accesses by other transactions.
- This is often shown in a "lock compatibility matrix":

Requested Lock	Existing Lock: None	Existing Lock: S	Existing Lock: X
S	+	+	−
X	+	−	−

Overview

Dirty Read Problem (1)

- Transaction A sets the balance of some account to 1 000 000, discovers the error, calls `ROLLBACK`.

Transaction A	Transaction B

`UPDATE Account`
`SET Balance = 1000000`
`WHERE Acc_No = 1`	

	`SELECT Balance`
	`FROM Account`
	`WHERE Acc_No = 1`
	$\longrightarrow$ 1 000 000

`ROLLBACK`	

Dirty Read Problem (2)

- So in the above schedule, Transaction B sees data which "never existed".
- Also if Transaction A later again changes the balance by another update, we would speak of a "Dirty Read" (even if Transaction A finally commits).
- No transaction should be able to observe an intermediate state of another transaction.
- Each transaction should see a transaction-consistent state, i.e. the result of a sequence of committed transactions (plus its own changes).

Dirty Read Problem (3)

- It is not difficult to exclude dirty reads, and most DBMS automatically do that.
- One possibility is to set write locks on tuples changed by a transaction and to keep these locks until the commit.

  More precisely, the locks are set before the change and removed after the commit (when the log entries are successfully written to the disk). Therefore, the uncommitted data is not accessible for other transactions.
- A transaction which wants to read a tuple tries to acquire a read lock for it. This is only possible if there is no write lock.

  The read lock can be taken off immediately after the read is processed (at least this is sufficient to avoid dirty reads).

Dirty Read Problem (4)

- Oracle instead uses "Multi Version Concurrency Control". For read accesses, it will restore versions of the accessed DB blocks which correspond to a state after the last committed transaction.
- I.e. all uncommitted changes are undone (based on information in the rollback segments which correspond to the undo part of the log) for the purpose of this read only.
- Also committed changes are undone if the commit was after the evaluation of the query started.

  So even for long-running SELECT commands, Oracle can guarantee that they will see a consistent state.

Dirty Read Problem (5)

Transaction A	Transaction B
SELECT Sal FROM Emp WHERE EName = 'JAMES' ⟶ 950	
	UPDATE Emp SET Sal = Sal * 2 WHERE EName = 'JAMES'
	SELECT ... ⟶ 1900
SELECT ... ⟶ 950	
	COMMIT
SELECT ... ⟶ 1900	

Lost Update (1)

- Consider two updates running concurrently.

  Before they are executed, the salary of James is 950.

Transaction A	Transaction B
UPDATE Emp SET Sal = Sal * 2 WHERE EName = 'JAMES'	UPDATE Emp SET Sal = Sal + 50 WHERE EName = 'JAMES'

- In a serial execution of the two transactions, the two possible outcomes are 1950 (A is executed before B), and 2000 (B is executed before A).
- However, each update consists of a read and a write, and it is dangerous to interleave the execution of these operations.

Lost Update (2)

- Without precautions, there are schedules of read and write operations in which only one of the two updates persists:

Transaction A	Transaction B
	`read(X, ’Sal ...’);`
`read(Y, ’Sal ...’);`	
`Y := Y * 2;`	
`write(Y, ’Sal ...’);`	
	`X := X + 50;`
	`write(X, ’Sal ...’);`

- So the second `write` overwrites the first, and the final salary in the database is only 1000.

Lost Update (3)

- Such “lost updates” are prevented by most DBMS.
- Oracle would acquire a write lock on the tuple for the employee “James” before it reads the value to be updated.
- So if Transaction B is the first to get this lock, then Transaction A has to wait with the update until B commits.

  This also avoids dirty reads. If we only want to avoid lost updates, B could free the lock immediately after the write.
- A DBMS also has to protect its internal operations by locks on blocks which it keeps only for a short time (“latches”).

  E.g. for an insertion, the inserted tuple is locked until the commit, but the block will be locked while the tuple is inserted (and data is moved within the block).

Lost Update (4)

- However, lost updates are automatically prevented only if you use an `UPDATE` command.
- If you need a more complex computation, and you first read the value with a `SELECT` command, and then write the new value with an `UPDATE` command, lost updates can happen.
- The problem is that a `SELECT` does normally not acquire a lock for the selected tuples (or only a short-living lock, which is removed immediately after the `SELECT`).
- Keeping locks on selected tuples until the end of the transaction would decrease concurrency/performance too much (and is often not necessary).

Lost Update (5)

Transaction A	Transaction B

`SELECT Sal FROM Emp`
`WHERE EName = ’JAMES’`
$\longrightarrow$ 950	

	`SELECT ...` $\longrightarrow$ 950

	`UPDATE Emp`
	`SET Sal = 1900`
	`WHERE EName = ’JAMES’`

	`COMMIT`

`UPDATE Emp`
`SET Sal = 1000`
`WHERE EName = ’JAMES’`	

`COMMIT`	

Lost Update (6)

- The above schedule with a lost update is possible in Oracle and other DBMS.
- In order to avoid it, you must add “FOR UPDATE” to the query. In this way, you tell the system that the SELECT result is in fact input for a later change.

```
SELECT Sal FROM Emp
WHERE EName = 'JAMES'
FOR UPDATE
```

- FOR UPDATE can only be used if the query would define an updatable view.

  Oracle allows joins, but no aggregations, DISTINCT, UNION.

Lost Update (7)

- The FOR UPDATE clause locks the tuples which satisfied the WHERE condition at the time of the query.
- The locks will be kept until the end of the transaction.
- You can also specify an attribute:

```
SELECT Sal FROM Emp
WHERE EName = 'JAMES'
FOR UPDATE OF Sal
```

- This is useful for systems which allow to lock single attributes.

  In Oracle, which allows joins in the queries, it also defines from which table rows should be locked.

Nonrepeatable Read (1)

- It is possible that you query the same data twice in a transaction, and you get different results:

Transaction A	Transaction B
SELECT Sal FROM Emp WHERE EName = 'JAMES' ⟶ 950	
	UPDATE Emp SET Sal = Sal + 50 WHERE EName = 'JAMES' COMMIT
SELECT Sal FROM Emp WHERE EName = 'JAMES' ⟶ 1000	

Nonrepeatable Read (2)

- This behaviour is not possible in a serial execution of transactions, so it violates the isolation principle.
- The same problem is the source of the second kind of lost update studied above.
- Of course, one probably would not pose exactly the same query twice in a transaction. But you might have queries which access overlapping sets of tuples.
- In a more general case, you might even access different tables. Again there is no guarantee that your two queries see the same database state (“inconsistent analysis problem”).

Nonrepeatable Read (3)

- E.g. you store the sum of all salaries redundantly in another table. If you first compute “`SELECT SUM(Sal) FROM Emp`” and then select the redundantly stored sum (e.g. in order to check the consistency), there is no guarantee that both queries are evaluated with respect to the same state.
- The “Nonrepeatable Read” can be avoided by keeping read locks on the accessed tuples until the end of the transaction.
- The “Inconsistent Analysis Problem” is harder.

  Oracle guarantees that each query is evaluated wrt only one state. So it would help to put the entire analysis into a single `SELECT` statement.

Phantom Problem (1)

- If we use only locks on tuples, an `INSERT` will never be prevented (non-existing tuples cannot be locked).
- Suppose the company has a budget of $5000 for a salary raise and that this money should be distributed evenly.

Transaction A	Transaction B
`SELECT COUNT(*) FROM Emp` $\longrightarrow$ 50	

	`INSERT INTO Emp`
	`VALUES (...)`
	`COMMIT`

`UPDATE Emp` `SET Sal = Sal + 100`	

Phantom Problem (2)

- So in this case, the new employee also got the salary raise, and $5100 were spent instead of the planned $5000.
- Note that it even had not help to use the query

```
SELECT * FROM Emp
FOR UPDATE OF Sal
```

  and to count the number of result rows.
- Even though now all existing rows of the table are locked, the insertion of new rows is not prevented.
- In this case you need to lock the entire table:

```
LOCK TABLE Emp IN EXCLUSIVE MODE
```


Phantom Problem (3)

- The `LOCK TABLE` command is not part of SQL-92.
- SQL-92 allows instead to specify an isolation level for your transaction (see below).

  This is also possible in Oracle, but see below.
- One proposal to avoid the phantom problem is to work with “Predicate Locks”. Instead of locking single tuples, you request that the set of tuples satisfying a condition cannot be changed.

  This, however, soon gives NP-complete problems.
- Special cases of such predicate locks can be processed efficiently by applying locks to indexes or other access paths.

Overview

1. Introduction
2. Problems
3. Final Remarks

Isolation Levels (1)

- SQL-92 (and Oracle) have a command

  `SET TRANSACTION ISOLATION LEVEL ⟨Level⟩`
- The SQL-Standard has four isolation levels.
- `READ UNCOMMITTED`: This means that the transaction can read the current DB state without waiting for locks.

  E.g. in order to compute the statistics for the optimizer, only an estimate is needed, and it doesn't matter whether some "dirty data" is contained in the statistics.
- `READ COMMITTED`: This is what was explained above. Read locks are only hold for the duration of the read.

Isolation Levels (2)

- `REPEATABLE READ`: Here read locks are only removed after the transaction committed.

  This does not protect against the phantom problem (and against an inconsistent analysis spanning multiple tables).
- `SERIALIZABLE`: This is the theoretical ideal of complete isolation. It also excludes the phantom problem.
- Oracle only supports "`READ COMMITTED`"/"`SERIALIZABLE`".
- The higher the isolation level you select, the less concurrency will be possible.
- In Oracle8, "`SERIALIZABLE`" is not really serializable, and it already allows only very little parallelism (see next slide).

Isolation Level "Serializable" in Oracle

- Suppose that there are two tables `R(A)` and `S(A)`, both with only one row containing the value `'old'`.
- Then the following schedule is possible in Oracle8, even after `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`.

Transaction A	Transaction B
`SELECT A FROM R` → `old`	
`UPDATE S SET A='new'`	
	`SELECT A FROM S` → `old`
	`UPDATE R SET A='new'`
	`COMMIT`
`COMMIT`	

Deadlocks (1)

- It is possible that two transactions wait on locks which are held by the other transaction:

Transaction A	Transaction B
`UPDATE Emp SET ...` `WHERE EName = 'JAMES'`	
	`UPDATE Emp SET ...` `WHERE EName = 'ALLEN'`
	`UPDATE Emp SET ...` `WHERE EName = 'JAMES'`
`UPDATE Emp SET ...` `WHERE EName = 'ALLEN'`	

Deadlocks (2)

- In this case, one of the transactions involved in the deadlock must be rolled back.

  This will free the locks hold by this transaction, and the other transaction can continue. Oracle does not roll back a transaction automatically, but will end one of the `UPDATE` requests with an error. In this case the application program should call `ROLLBACK`.
- The deadlock test is costly, therefore some systems do it only from time to time (or only after a transaction had waited some time for a lock).
- One should analyze the application programs for possible deadlocks.

Locks and Performance (1)

- Suppose you have the task to generate order numbers.
- You can use a table with a single row to store the last order number (which was already used).

OrderNo
Last_Used
25

- So then you could use the following commands to get the next order number:

```
UPDATE OrderNo SET Last_Used = Last_Used + 1;
SELECT Last_Used FROM OrderNo
```


Locks and Performance (2)

- The problem is that once a transaction has generated a new order number in this way, the only row of `OrderNo` is locked until this transaction commits.

  So we cannot enter orders in parallel into the database.
- Since this problem occurs often, Oracle has a special mechanism for generating unique numbers.
- Instead of using a table, you create a “sequence”:

  ```
  CREATE SEQUENCE ORDERNO;
  ```

  This command has many optional parameters, see the manual.
- Then `ORDERNO.NEXTVAL` increments the value and returns it:

  ```
  INSERT INTO ORDERS(ORDERNO.NEXTVAL, ...)
  ```

Locks and Performance (3)

- Suppose you want these two properties:
  - Sequence numbers should be increasing, i.e. if an order was entered later, then it should have a higher number.
  - There should be no holes in the used sequence numbers.
- Then, once a transaction A has requested a sequence number, you cannot give another sequence number to a transaction B until A commits.

  If A commits, B must get a higher number. But if A calls `ROLLBACK`, B must reuse this number to avoid holes.
- Therefore, Oracle allows holes in the generated numbers.

  With the parallel server, also the order is not guaranteed.