16

OVERVIEW OF TRANSACTION MANAGEMENT

Exercise 16.1 Give brief answers to the following questions:

1. What is a transaction? In what ways is it different from an ordinary program (in a language such as C)?
2. Define these terms: *atomicity, consistency, isolation, durability, schedule, blind write, dirty read, unrepeatable read, serializable schedule, recoverable schedule, avoids-cascading-aborts schedule.*
3. Describe Strict 2PL.
4. What is the phantom problem? Can it occur in a database where the set of database objects is fixed and only the values of objects can be changed?

Answer 16.1 The answer to each question is given below.

1. A *transaction* is an execution of a user program, and is seen by the DBMS as a series or list of actions. The actions that can be executed by a transaction include reads and writes of database objects, whereas actions in an ordinary program could involve user input, access to network devices, user interface drawing, etc.
2. Each term is described below.
   (a) *Atomicity* means a transaction executes when all actions of the transaction are completed fully, or none are. This means there are no partial transactions (such as when half the actions complete and the other half do not).
   (b) *Consistency* involves beginning a transaction with a 'consistent' database, and finishing with a 'consistent' database. For example, in a bank database, money should never be "created" or "deleted" without an appropriate deposit or withdrawal. Every transaction should see a consistent database.

(c) *Isolation* ensures that a transaction can run independently, without considering any side effects that other concurrently running transactions might have. When a database interleaves transaction actions for performance reasons, the database protects each transaction from the effects of other transactions.

(d) *Durability* defines the persistence of committed data: once a transaction commits, the data should persist in the database even if the system crashes before the data is written to non-volatile storage.

(e) A *schedule* is a series of (possibly overlapping) transactions.

(f) A *blind write* is when a transaction writes to an object without ever reading the object.

(g) A *dirty read* occurs when a transaction reads a database object that has been modified by another not-yet-committed transaction.

(h) An *unrepeatable read* occurs when a transaction is unable to read the same object value more than once, even though the transaction has not modified the value. Suppose a transaction T2 changes the value of an object A that has been read by a transaction T1 while T1 is still in progress. If T1 tries to read the value of A again, it will get a different result, even though it has not modified A.

(i) A *serializable schedule* over a set S of transactions is a schedule whose effect on any consistent database instance is identical to that of some complete serial schedule over the set of committed transactions in S.

(j) A *recoverable schedule* is one in which a transaction can commit only after all other transactions whose changes it has read have committed.

(k) A schedule that *avoids-cascading-aborts* is one in which transactions only read the changes of committed transactions. Such a schedule is not only recoverable, aborting a transaction can be accomplished without cascading the abort to other transactions.

3. *Strict 2PL* is the most widely used locking protocol where 1) A transaction requests a shared/exclusive lock on the object before it reads/modifies the object. 2) All locks held by a transaction are released when the transaction is completed.

4. The *phantom problem* is a situation where a transaction retrieves a collection of objects twice but sees different results, even though it does not modify any of these objects itself and follows the strict 2PL protocol. This problem usually arises in dynamic databases where a transaction cannot assume it has locked all objects of a given type (such as all sailors with rank 1; new sailors of rank 1 can be added by a second transaction after one transaction has locked all of the original ones).

   If the set of database objects is fixed and only the values of objects can be changed, the phantom problem cannot occur since one cannot insert new objects into the database.

Exercise 16.2 Consider the following actions taken by transaction $T1$ on database objects X and Y:

R(X), W(X), R(Y), W(Y)

1. Give an example of another transaction $T2$ that, if run concurrently to transaction T without some form of concurrency control, could interfere with $T1$.
2. Explain how the use of Strict 2PL would prevent interference between the two transactions.
3. Strict 2PL is used in many database systems. Give two reasons for its popularity.

Answer 16.2 The answer to each question is given below.

1. If the transaction T2 performed $W(Y)$ before T1 performed $R(Y)$, and then T2 aborted, the value read by T1 would be invalid and the abort would be cascaded to T1 (i.e. T1 would also have to abort).
2. Strict 2PL would require T2 to obtain an exclusive lock on Y before writing to it. This lock would have to be held until T2 committed or aborted; this would block T1 from reading Y until T2 was finished, thus there would be no interference.
3. Strict 2PL is popular for many reasons. One reason is that it ensures only 'safe' interleaving of transactions so that transactions are recoverable, avoid cascading aborts, etc. Another reason is that strict 2PL is very simple and easy to implement. The lock manager only needs to provide a lookup for exclusive locks and an atomic locking mechanism (such as with a semaphore).

Exercise 16.3 Consider a database with objects X and Y and assume that there are two transactions $T1$ and $T2$. Transaction $T1$ reads objects X and Y and then writes object X. Transaction $T2$ reads objects X and Y and then writes objects X and Y.

1. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a write-read conflict.
2. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a read-write conflict.
3. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a write-write conflict.
4. For each of the three schedules, show that Strict 2PL disallows the schedule.

Answer 16.3 The answer to each question is given below.

1. The following schedule results in a write-read conflict:
   T2:R(X), T2:R(Y), T2:W(X), T1:R(X) ...
   T1:R(X) is a dirty read here.

2. The following schedule results in a read-write conflict:
   T2:R(X), T2:R(Y), T1:R(X), T1:R(Y), T1:W(X) ...
   Now, T2 will get an unrepeatable read on X.

3. The following schedule results in a write-write conflict:
   T2:R(X), T2:R(Y), T1:R(X), T1:R(Y), T1:W(X), T2:W(X) ...
   Now, T2 has overwritten uncommitted data.

4. Strict 2PL resolves these conflicts as follows:

   (a) In S2PL, T1 could not get a shared lock on X because T2 would be holding an exclusive lock on X. Thus, T1 would have to wait until T2 was finished.

   (b) Here T1 could not get an exclusive lock on X because T2 would already be holding a shared or exclusive lock on X.

   (c) Same as above.

Exercise 16.4 We call a transaction that only reads database object a **read-only** transaction, otherwise the transaction is called a **read-write** transaction. Give brief answers to the following questions:

1. What is lock thrashing and when does it occur?

2. What happens to the database system throughput if the number of read-write transactions is increased?

3. What happens to the database system throughput if the number of read-only transactions is increased?

4. Describe three ways of tuning your system to increase transaction throughput.

Answer 16.4 The answer to each question is given below.

1. *Locking thrashing* occurs when the database system reaches to a point where adding another new active transaction actually reduces throughput due to competition for locking among all active transactions. Empirically, locking thrashing is seen to occur when 30% of active transactions are blocked.

2. If the number of read-write transaction is increased, the database system throughput will increase until it reaches the thrashing point; then it will decrease since read-write transactions require exclusive locks, thus resulting in less concurrent execution.

3. If the number of read-only transaction is increased, the database system throughput will also increase since read-only transactions require only shared locks. So we are able to have more concurrency and execute more transactions in a given time.

4. Throughput can be increased in three ways:

   (a) By locking the smallest sized objects possible.

   (b) By reducing the time that transaction hold locks.

   (c) By reducing hot spots, a database object that is frequently accessed and modified.

Exercise 16.5 Suppose that a DBMS recognizes *increment*, which increments an integer-valued object by 1, and *decrement* as actions, in addition to reads and writes. A transaction that increments an object need not know the value of the object; increment and decrement are versions of blind writes. In addition to shared and exclusive locks, two special locks are supported: An object must be locked in I mode before incrementing it and locked in D mode before decrementing it. An I lock is compatible with another I or D lock on the same object, but not with S and X locks.

1. Illustrate how the use of I and D locks can increase concurrency. (Show a schedule allowed by Strict 2PL that only uses S and X locks. Explain how the use of I and D locks can allow more actions to be interleaved, while continuing to follow Strict 2PL.)

2. Informally explain how Strict 2PL guarantees serializability even in the presence of I and D locks. (Identify which pairs of actions conflict, in the sense that their relative order can affect the result, and show that the use of S, X, I, and D locks according to Strict 2PL orders all conflicting pairs of actions to be the same as the order in some serial schedule.)

Answer 16.5 The answer to each question is given below.

1. Take the following two transactions as example:

   T1: Increment A, Decrement B, Read C;
   T2: Increment B, Decrement A, Read C

   If using only strict 2PL, all actions are versions of blind writes, they have to obtain exclusive locks on objects. Following strict 2PL, T1 gets an exclusive lock on A, if T2 now gets an exclusive lock on B, there will be a deadlock. Even if T1 is fast enough to have grabbed an exclusive lock on B first, T2 will now be blocked until T1 finishes. This has little concurrency. If I and D locks are used, since I and

D are compatible, T1 obtains an I-Lock on A, and a D-Lock on B; T2 can still obtain an I-Lock on B, a D-Lock on A; both transactions can be interleaved to allow maximum concurrency.

2. The pairs of actions which conflicts are:

   RW, WW, WR, IR, IW, DR, DW

   We know that strict 2PL orders the first 3 conflicts pairs of actions to be the same as the order in some serial schedule. We can also show that even in the presence of I and D locks, strict 2PL also orders the latter 4 pairs of actions to be the same as the order in some serial schedule. Think of an I (or D)lock under these circumstances as an exclusive lock, since an I(D) lock is not compatible with S and X locks anyway (ie. can't get a S or X lock if another transaction has an I or D lock). So serializability is guaranteed.

Exercise 16.6 Answer the following questions: SQL supports four isolation-levels and two access-modes, for a total of eight combinations of isolation-level and access-mode. Each combination implicitly defines a class of transactions; the following questions refer to these eight classes:

1. Consider the four SQL isolation levels. Describe which of the phenomena can occur at each of these isolation levels: *dirty read, unrepeatable read, phantom problem.*
2. For each of the four isolation levels, give examples of transactions that could be run safely at that level.
3. Why does the access mode of a transaction matter?

Answer 16.6 The answer to each question is given below.

1.

Level	Dirty Read	Unrepeatable read	phantom problem
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

2. (a) A SERIALIZABLE transaction achieves the highest degree of isolation from the effects of other transactions. It obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged and hold them until the end. Thus it is immune to all three phenomena above. We can safely run transactions like (assuming T2 inserts new objects):
   T1: R(X), R(Y), W(X). W(Y) Commit
   T2: W(X), W(Y) Commit

(b) A REPEATABLE READ transaction sets the same locks as a SERIALIZABLE transaction, except that it locks only objects, not sets of objects. We can safely run transactions like (assuming T1 or T2 does not insert any new objects):
T1: R(X), R(Y), W(X), W(Y), Commit
T2: R(X), W(X), R(Y), W(Y), Commit

(c) A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared lock before reading, but it releases it immediately. Thus it is only immune to dirty read. So we can safely run transactions like (assuming T1 or T2 does not insert any new objects):
T1:R(X), W(X), Commit
T2:R(X), W(X), Commit

(d) A READ UNCOMMITTED transaction can never make any lock requests, thus it is vulnerable to dirty read, unrepeatable read and phantom problem. Transactions which run safely at this level can only read objects from the database:
T1:R(X), R(Y), Commit
T2:R(Y), R(X), Commit

3. Access mode of a transaction tells what kind of lock is needed by the transaction. If the transaction is with READ ONLY access mode, only shared locks need to be obtained, thereby increases concurrency.

Exercise 16.7 Consider the university enrollment database schema:

Student(*snum:* `integer`, *sname:* `string`, *major:* `string`, *level:* `string`, *age:* `integer`)
Class(*name:* `string`, *meets_at:* `time`, *room:* `string`, *fid:* `integer`)
Enrolled(*snum:* `integer`, *cname:* `string`)
Faculty(*fid*: `integer`, *fname:* `string`, *deptid:* `integer`)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

For each of the following transactions, state the SQL isolation level you would use and explain why you chose it.

1. Enroll a student identified by her *snum* into the class named 'Introduction to Database Systems'.

2. Change enrollment for a student identified by her *snum* from one class to another class.
3. Assign a new faculty member identified by his *fid* to the class with the least number of students.
4. For each class, show the number of students enrolled in the class.

Answer 16.7 The answer to each question is given below.

1. Because we are inserting a new row in the table *Enrolled*, we do not need any lock on the existing rows. So we would use READ UNCOMMITTED.
2. Because we are updating one existing row in the table *Enrolled*, we need an exclusive lock on the row which we are updating. So we would use READ COMMITTED.
3. To prevent other transactions from inserting or updating the table *Enrolled* while we are reading from it (known as the phantom problem), we would need to use SERIALIZABLE.
4. same as above.

Exercise 16.8 Consider the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers.

For each of the following transactions, state the SQL isolation level that you would use and explain why you chose it.

1. A transaction that adds a new part to a supplier's catalog.
2. A transaction that increases the price that a supplier charges for a part.
3. A transaction that determines the total number of items for a given supplier.
4. A transaction that shows, for each part, the supplier that supplies the part at the lowest price.

Answer 16.8 The answer to each question is given below.

1. Because we are inserting a new row in the table *Catalog*, we do not need any lock on the existing rows. So we would use READ UNCOMMITTED.

2. Because we are updating one existing row in the table *Catalog*, we need an exclusive lock on the row which we are updating. So we would use READ COMMITTED.

3. To prevent other transactions from inserting or updating the table *Catalog* while we are reading from it (known as the phantom problem), we would need to use SERIALIZABLE.

4. same as above.

Exercise 16.9 Consider a database with the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers.

Consider the transactions $T1$ and $T2$. $T1$ always has SQL isolation level `SERIALIZABLE`. We first run $T1$ concurrently with $T2$ and then we run $T1$ concurrently with $T2$ but we change the isolation level of $T2$ as specified below. Give a database instance and SQL statements for $T1$ and $T2$ such that result of running $T2$ with the first SQL isolation level is different from running $T2$ with the second SQL isolation level. Also specify the common schedule of $T1$ and $T2$ and explain why the results are different.

1. `SERIALIZABLE` versus `REPEATABLE READ`.

2. `REPEATABLE READ` versus `READ COMMITTED`.

3. `READ COMMITTED` versus `READ UNCOMMITTED`.

Answer 16.9 The answer to each question is given below.

1. Suppose a database instance of table Catalog and SQL statements shown below:

sid	pid	cost
18	45	$7.05
22	98	$89.35
31	52	$357.65
31	53	$26.22
58	15	$37.50
58	94	$26.22