Atomicity, consistency, independence and durablity(ACID) principle

Hi! In this chapter I am going to discuss with you about ACID property.

INTRODUCTION

Most modern computer systems, except the personal computers and many workstations, are multiuser systems. Mulitple users are able to use a single process simultaneously because of multiprogramming in which the processor (or processors) in the system is shared amongst a number of processes trying to access computer resources (including databases) simultaneously. The concurrent execution of programs is therefore interleaved with each program being allowed access to the CPU at regular intervals. This also enables another program to access the CPU while a program is doing I/O. In this chapter we discuss the problem of synchronization of access to shared objects in a database while supporting a high degree of concurrency.

Concurrency control in database management systems permits many users (assumed to be interactive) to access a database in a multiprogrammed environment while preserving the illusion that each user has sole access to the system. Control is needed to coordinate concurrent accesses to a DBMS so that the overall correctness of the database is maintained. Efficiency is also important since the response time for interactive users ought to be short. The reader may have recognised that the concurrency problems in database management systems are related to the concurrency problems in operating systems although there are significant differences between the two. For example, operating systems only involve concurrent sharing of resources while the DBMS must deal with a number of users attempting to concurrently access and modify data in the database.

Clearly no problem arises if all users were accessing the database only to retrieve information and no one was modifying it, for example, accessing the census data or a library catalogue. If one or more of the users were modifying the database e.g. a bank account or airline reservations, an update performed by one user may interfere with an

update or retrieval by another user. For example, users *A* and *B* both may wish to read and update the same record in the database at about the same time. The relative timing of the two transactions may have an impact on the state of the database at the end of the transactions. The end result may be an inconsistent database.

Our discussion of concurrency will be transaction based. A transaction is a sequence of actions $[t_1, t_2, \ldots, t_n]$. As noted in the last chapter, a transaction is a unit of consistency in that it preserves database consistency. We assume that all transactions complete successfully; problems of transactions failures are resolved by the recovery mechanisms. The only detail of transactions that interests us right now is their reads and writes although other computation would often be carried out between the reads and writes. We therefore assume that all actions t_i that form a transaction are either a read or a write. The set of items read by a transaction are called its *read set* and the set of items written by it are called its *write set*. Two transactions T_i and T_j are said to *conflict* if some action t_i of T_i and an action t_j of T_j access the same object and at least one of the actions is a write. Two situations are possible:

1. The write set of one transaction intersects with the read set of another. The result of running the two transactions concurrently will clearly depend on whether the write is done first or the read is. The conflict is called a *RW-conflict*;
2. The write set of one transaction intersects with the write set of another. Again, the result of running the two transactions concurrently will depend on the order of the two writes. The conflict is called a *WW-conflict*.

A concurrency control mechanism must detect such conflicts and control them. Various concurrency control mechanisms are available. The mechanisms differ in the time they detect the conflict and the way they resolve it. We will consider the control algorithms later. First we discuss some examples of concurrency anomalies to highlight the need for concurrency control.

We have noted already that in the discussion that follows we will ignore many details of a transaction. For example, we will not be concerned with any computations other than the READs and the WRITEs and whether results of a READ are being stored in a local variable or not.

ACID properties

ACID properties are an important concept for databases. The acronym stands for Atomicity, Consistency, Isolation, and Durability.

The ACID properties of a DBMS allow safe sharing of data. Without these ACID properties, everyday occurrences such using computer systems to buy products would be difficult and the potential for inaccuracy would be huge. Imagine more than one person trying to buy the same size and color of a sweater at the same time -- a regular occurrence. The ACID properties make it possible for the merchant to keep these sweater purchasing transactions from overlapping each other -- saving the merchant from erroneous inventory and account balances.

Atomicity

The phrase "all or nothing" succinctly describes the first ACID property of atomicity. When an update occurs to a database, either all or none of the update becomes available to anyone beyond the user or application performing the update. This update to the database is called a transaction and it either commits or aborts. This means that only a fragment of the update cannot be placed into the database, should a problem occur with either the hardware or the software involved. Features to consider for atomicity:

a transaction is a unit of operation - either all the transaction's actions are completed or none are

- atomicity is maintained in the presence of deadlocks
- atomicity is maintained in the presence of database software failures
- atomicity is maintained in the presence of application software failures
- atomicity is maintained in the presence of CPU failures

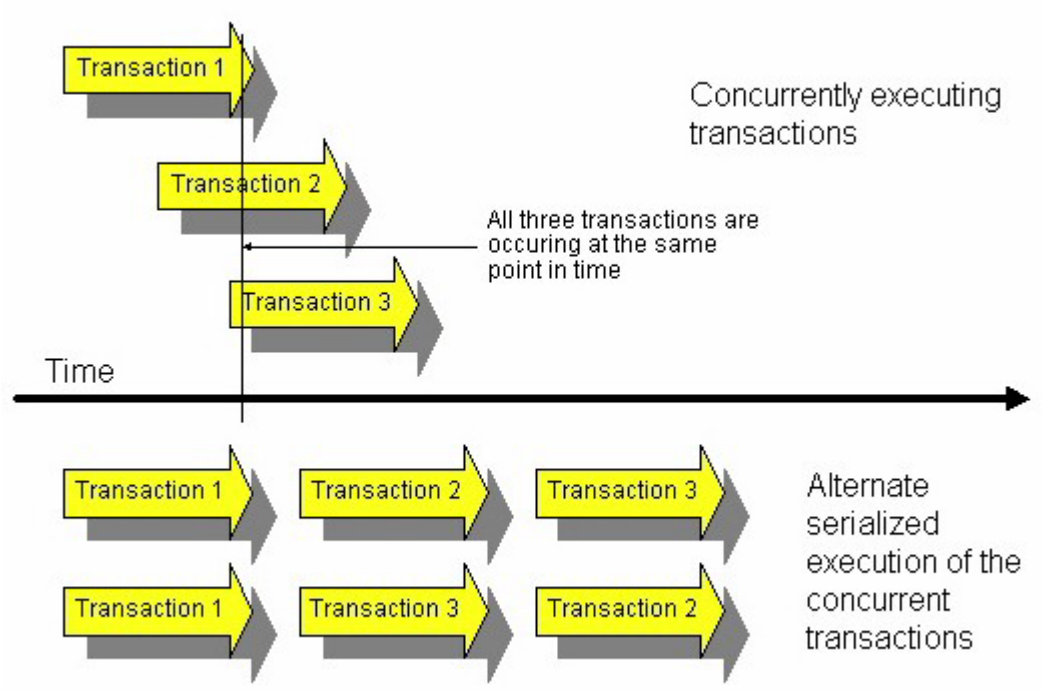
- atomicity is maintained in the presence of disk failures
- atomicity can be turned off at the system level
- atomicity can be turned off at the session level

Consistency

Consistency is the ACID property that ensures that any changes to values in an instance are consistent with changes to other values in the same instance. A consistency constraint is a predicate on data which server as a precondition, post-condition, and transformation condition on any transaction

Isolation

The isolation portion of the ACID properties is needed when there are concurrent transactions. Concurrent transactions are transactions that occur at the same time, such as shared multiple users accessing shared objects. This situation is illustrated at the top of the figure as activities occurring over time. The safeguards used by a DBMS to prevent conflicts between concurrent transactions are a concept referred to as isolation.



As an example, if two people are updating the same catalog item, it's not acceptable for one person's changes to be "clobbered" when the second person saves a different set of changes. Both users should be able to work in isolation, working as though he or she is the only user. Each set of changes must be isolated from those of the other users.

An important concept to understanding isolation through transactions is serializability. Transactions are serializable when the effect on the database is the same whether the transactions are executed in serial order or in an interleaved fashion. As you can see at the top of the figure, Transactions 1 through Transaction 3 are executing concurrently over time. The effect on the DBMS is that the transactions may execute in serial order based on consistency and isolation requirements. If you look at the bottom of the figure, you can see several ways in which these transactions may execute. It is important to note that a serialized execution does not imply the first transactions will automatically be the ones that will terminate before other transactions in the serial order.

Degrees of isolation[1]:

- degree 0 - a transaction does not overwrite data updated by another user or process ("dirty data") of other transactions
- degree 1 - degree 0 plus a transaction does not commit any writes until it completes all its writes (until the end of transaction)
- degree 2 - degree 1 plus a transaction does not read dirty data from other transactions
- degree 3 - degree 2 plus other transactions do not dirty data read by a transaction before the transaction commits

[1] These were originally described as *degrees of consistency* by Jim Gray. The following book provides excellent, updated coverage of the concept of isolation along with other transaction concepts

Durability

Maintaining updates of committed transactions is critical. These updates must never be lost. The ACID property of durability addresses this need. Durability refers to the ability of the system to recover committed transaction updates if either the system or the storage media fails. Features to consider for durability:

- recovery to the most recent successful commit after a database software failure
- recovery to the most recent successful commit after an application software failure
- recovery to the most recent successful commit after a CPU failure
- recovery to the most recent successful backup after a disk failure
- recovery to the most recent successful commit after a data disk failure

---

- Examples of Concurrency Anomalies
    - Lost Updates
    - Inconsistent Retrievals
    - Uncommitted Dependency

Examples of Concurrency Anomalies

There are three classical concurrency anomalies. These are lost updates, inconsistent retrievals and uncommitted dependency.

- Lost Updates
- Inconsistent Retrievals
- Uncommitted Dependency

Lost Updates

Suppose two users *A* and *B* simultaneously access an airline database wishing to reserve a number of seats on a flight. Let us assume that *A* wishes to reserve five seats while *B*

wishes to reserve four seats. The reservation of seats involves booking the seats and then updating the number of seats available (*N*) on the flight. The two users read-in the present number of seats, modify it and write back resulting in the following situation:

A	Time	B
Read N	1	-
-	2	Read N
N := N-5	3	-
-	4	N := N-4
Write N	5	-
-	6	Write N

Figure 1 An Example of Lost Update Anomaly

If the execution of the two transactions were to take place as shown in Fugure 1, the update by transaction *A* is lost although both users read the number of seats, get the bookings confirmed and write the updated number back to the database. This is called the *lost update anomaly* since the effects of one of the transactions were lost.

Inconsistent Retrievals

Consider two users *A* and *B* accessing a department database simultaneously. The user *A* is updating the database to give all employees in the department a 5% raise in their salary while the other user wants to know the total salary bill of the department. The two transactions interfere since the total salary bill would be changing as the first user updates the employee records. The total salary retrieved by the second user may be a sum of some salaries before the raise and others after the raise. This could not be considered an acceptable value of the total salary but the value before the raise or the value after the raise is acceptable.

A	Time	B
Read Employee 100	1	-
-	2	Sum = 0.0
Update Salary	3	-
-	4	Read Employee 100
Write Employee 100	5	-
-	6	Sum = Sum + Salary
Read Employee 101	7	-
-	8	-
Update Salary	9	-
Write Employee 101	10	-
-	11	-
-	12	Read Employee 101
-	13	Sum = Sum + Salary
etc	-	etc

Figure 2 An Example of Inconsistent Retrieval

The problem illustrated in the last example is called the *inconsistent retrieval anomaly*. During the execution of a transaction therefore, changes made by another transaction that has not yet committed should not be visible since that data may not be consistent.

Uncommitted Dependency

In the last chapter on recovery we discussed a recovery technique that involves immediate updates of the database and the maintenance of a log for recovery by rolling back the transaction in case of a system crash or transaction failure. If this technique was being used, we could have the following situation:

A	Time	B
-	1	Read Q
-	2	-
-	3	Write Q
Read Q	4	-
-	5	Read R
-	6	-
Write Q	7	-
-	8	Failure (rollback)
-	9	-

Figure 3 An Example of Uncommitted Dependency

Transaction *A* has now read the value of *Q* that was updated by transaction *B* but was never committed. The result of Transaction *A* writing *Q* therefore will lead to an inconsistent state of the database. Also if the transaction *A* doesn't write *Q* but only reads it, it would be using a value of *Q* which never really existed! Yet another situation would occur if the roll back happens after *Q* is written by transaction *A*. The roll back would restore the old value of *Q* and therefore lead to the loss of updated *Q* by transaction *A*. This is called the *uncommitted dependency anomaly*.

We will not discuss the problem of uncommitted dependency any further since we assume that the recovery algorithm will ensure that transaction *A* is also rolled back when *B* is. The most serious problem facing concurrency control is that of the lost update. The reader is probably already thinking of some solutions to the problem. The commonly suggested solutions are:

1. **Once transaction *A* reads a record *Q* for an update, no other transaction is allowed to read it until transaction *A* update is completed. This is usually called *locking*.**
2. **Both transaction *A* and *B* are allowed to read the same record *Q* but once *A* has updated the record, *B* is not allowed to update it as well since *B* would**

now be updating an old copy of the record *Q*. *B* must therefore read the record again and then perform the update.

3. **Although both transactions *A* and *B* are allowed to read the same record *Q*, *A* is not allowed to update the record because another transaction (Transaction *B*) has the old value of the record.**
4. **Partition the database into several parts and schedule concurrent transactions such that each transaction uses a different partition. There is thus no conflict and database stays consistent. Often this is not feasible since most databases have some parts (called *hot spots*) that most transactions want to access.**

These are in fact the major concurrency control techniques. We discuss them in detail later in this chapter. We first need to consider the concept of *serializability* which deals with correct execution of transactions concurrently. [phantoms???]

Review Question

1. Define ACID Property

Selected Bibliography

- [ARIES] C. Mohan, et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging., TODS 17(1): 94-162 (1992).
- [CACHE] C. Mohan: Caching Technologies for Web Applications, A Tutorial at the Conference on Very Large Databases (VLDB), Rome, Italy, 2001.
- [CODASYL] ACM: CODASYL Data Base Task Group April 71 Report, New York, 1971.
- [CODD] E. Codd: A Relational Model of Data for Large Shared Data Banks. ACM 13(6):377-387 (1970).

- [EBXML] http://www.ebxml.org.
- [FED] J. Melton, J. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, K. Zeidenstein: SQL and Management of External Data', SIGMOD Record 30(1):70-77, 2001.
- [GRAY] Gray, et al.: Granularity of Locks and Degrees of Consistency in a Shared Database., IFIP Working Conference on Modelling of Database Management Systems, 1-29, AFIPS Press.
- [INFO] P. Lyman, H. Varian, A. Dunn, A. Strygin, K. Swearingen: How Much Information? at http://www.sims.berkeley.edu/research/projects/how-much-info/.
- [LIND] B. Lindsay, et. al: Notes on Distributed Database Systems. IBM Research Report RJ2571, (1979).