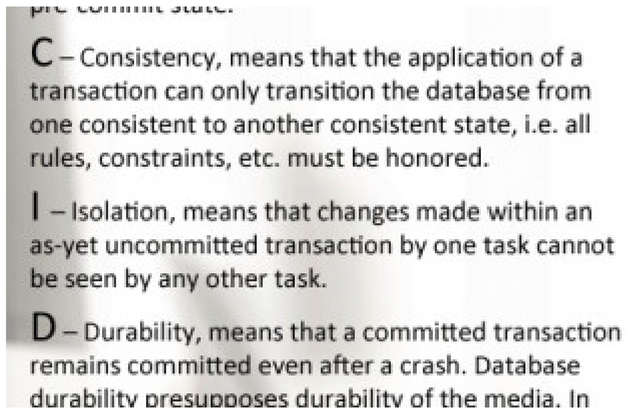
Comparing Optimistic and Pessimistic Concurrency

By **Steve Graves**
CEO
MCOBJECT

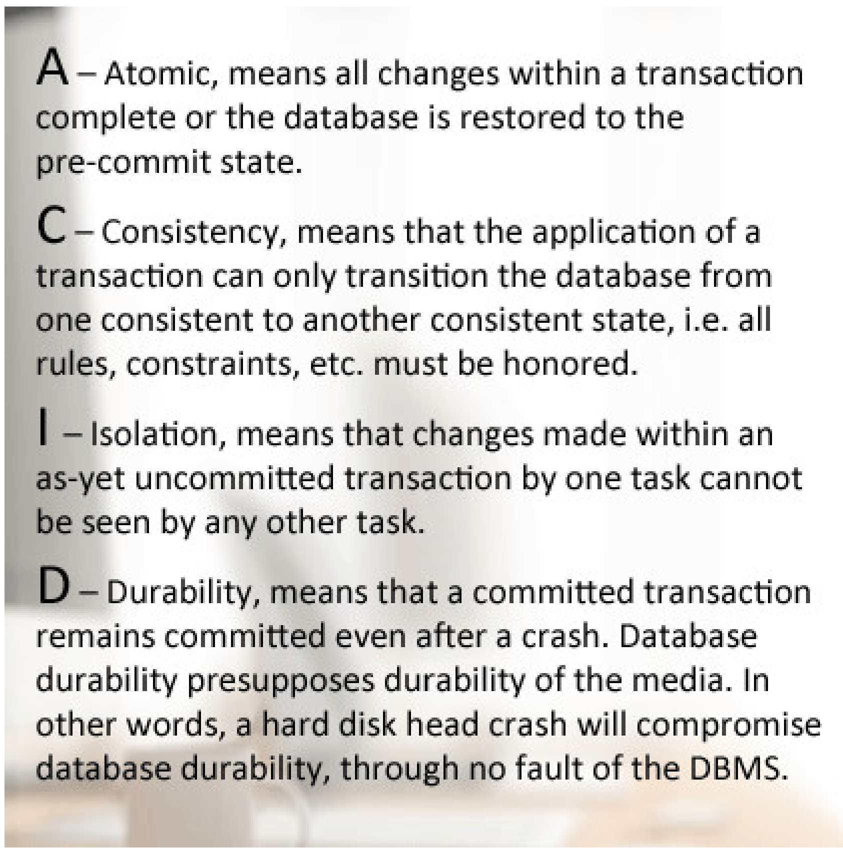
June 02, 2020

STORY

Usually (with rare exception), a database is a shared resource, meaning that it is used concurrently by two or more tasks.



Usually (with rare exception), a database is a shared resource, meaning that it is used concurrently by two or more tasks. This leads us to the topic of concurrency control; i.e. how do we coordinate a tasks' access to the database? This is part-and-parcel of providing a database management system that adheres to the ACID properties; see image #1.



A – Atomic, means all changes within a transaction complete or the database is restored to the pre-commit state.

C – Consistency, means that the application of a transaction can only transition the database from one consistent to another consistent state, i.e. all rules, constraints, etc. must be honored.

I – Isolation, means that changes made within an as-yet uncommitted transaction by one task cannot be seen by any other task.

D – Durability, means that a committed transaction remains committed even after a crash. Database durability presupposes durability of the media. In other words, a hard disk head crash will compromise database durability, through no fault of the DBMS.

The isolation property is implemented through concurrency control. Broadly speaking, there are two forms of concurrency control: pessimistic and optimistic.

These are so called because pessimistic concurrency control proactively prevents harm (harm, in this case, being a violation of the isolation property), whereas optimistic concurrency control assumes that no harm will happen, but if it is detected only then will measures be taken.

In operational terms, pessimistic concurrency control is affected with locks. Different vendors can, and do, offer different levels of locks: database locks, table locks, row locks and even column locks in extreme cases. For locking at any level of granularity below the database, things get complicated: there must be either a mechanism to prevent deadlocks (aka race conditions), or a means to detect and remediate deadlocks. Invariably, this involves a separate process called a lock manager or lock arbiter. This is problematic when a database system is supposed to be an 'embedded database'. Ideally, an embedded database is entirely contained (embedded) within the application.

A deadlock happens when:

Process 1	Process 2
Holds a lock on 'A'	Holds a lock on 'B'
Requests a lock on 'B'	Requests a lock on 'A'

Deadlocks can be prevented by grouping lock requests, i.e. all the locks that will be required within the scope of a transaction are acquired at once. This works for table level locks but is rarely possible for row level locks.

For pessimistic concurrency control the database is locked for the exclusive use of a task with a READ_WRITE transaction. This eliminates the need for a lock manager/arbiter or to deal with deadlock prevention or detection, and greatly simplifies the transaction manager. eXtremeDB and SQLite are two examples of embedded databases that provide database locking. Given the speed of an in-memory database, this coarse level of locking is well-justified: the time (CPU cycles) that would be required for complex lock arbitration would exceed the time a process needs to simply get into the database, do its work, and get out. This calculus also holds up for persistent databases that have a high ratio of READ_ONLY to READ_WRITE transactions, and there are few concurrent READ_WRITE transactions at any given moment, and transactions are short-lived. The definition of "few" and "short-lived" varies for relatively fast SSD compared to relatively slow HDD.

With database locking, the concurrent access pattern looks like image #2.

	Process 1	Process 2
Epoch 1	Start READ_WRITE	
Epoch 2		Start READ_WRITE
Epoch 3	Read record A	Waiting
Epoch 4	Modify field A.b	Waiting
Epoch 5	Write record A	Waiting
Epoch 6	Commit	Waiting
Epoch 7	Start READ_ONLY	Read record A
Epoch 8	Waiting	Modify field A.c
Epoch 9	Waiting	Modify field A.d
Epoch 10	Waiting	Write record A
Epoch 11	Waiting	Commit
Epoch 12	Start reading...	Start READ_ONLY
Epoch 13	Reading	Start reading
Epoch 14	Reading	Reading
Epoch 15	Commit	Commit

So, database locking prevents harm, i.e. the violation of the isolation property, by blocking any task from accessing the database while another task is modifying the database. The other tasks will wait until the database system can schedule their access to the database. In an environment with many tasks that frequently modify the database, having all but one of them blocked at any given moment is undesirable, so optimistic concurrency control could be advantageous.

Optimistic concurrency control is most often implemented with a MVCC (Multi-Version Concurrency Control) implementation. The optimistic model doesn't block tasks from concurrent access to the database. This approach optimistically assumes that tasks will not violate each other's isolation, e.g. that scenario depicted in image #3 won't happen.

	Process 1	Process 2
Epoch 1	Read record A	Read record A
Epoch 2	Modify field A.b	Modify field A.c

Epoch 3	Write record A	Modify field A.d
Epoch 4		Write record A

In this scenario, when Process 2 writes record A back to the database, it reverts field A.b to the value when both Process 1 and 2 read the record, obliterating Process 1's change to field A.b. This scenario is generically called a conflict.

But we can't guarantee that this won't happen, so MVCC needs a mechanism to detect when a conflict would happen if a transaction was allowed to commit. This is done by creating versions of the record in the database (hence the name, Multi-Version Concurrency Control), and then checking the version number when the transaction is committed. The above scenario becomes as shown in image #4.

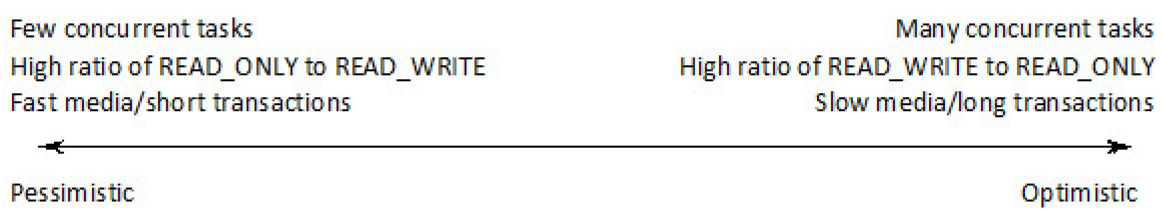
	Process 1	Process 2
Epoch 1	Start READ_WRITE	Start READ_WRITE
Epoch 2	Read record A_0	Read record A_0
	MVCC gives copy of A_0	MVCC gives copy of A_0
Epoch 3	Modify field A_0.b	Modify field A_0.c
Epoch 4	Write record A_0	Modify field A_0.d
Epoch 5	Commit	
Epoch 6	MVCC creates A_1	
Epoch 5		Write record A_0
Epoch 6		Commit
		A_0 is older than A_1, CONFLICT!!! (Commit fails)
		Retry the transaction from the start

Obviously, there is some overhead involved with MVCC. Versions (copies) have to be made of records and those versions checked prior to allowing the transaction to commit, and in the event of a conflict, the transaction has to be executed again from the beginning, and finally, old versions have to be cleaned up when they're no longer in use. So, there are assumptions underlying MVCC: First, that the overhead associated with versions is comparable to the overhead of managing locks, deadlock detection, etc. associated with pessimistic concurrency control, that the benefit of not having tasks blocked (i.e. greater concurrency) justifies slightly more complex programming (to handle possible conflicts) and that conflicts will be rare. If conflicts are common and transactions are being re-executed frequently, then a pessimistic concurrency control solution might be a better alternative.

In the absence of conflicts, MVCC exhibits far superior concurrency. See https://www.mcobject.com/mvcc#performance

The table in image #5 summarizes the optimistic and pessimistic comparison, which can also be imagined as the continuum illustrated image #6.

Characteristic	**MURSIW**	**MVCC**
One or few concurrent writers	Great	Good
High ratio of READ_ONLY to READ_WRITE	Great	Good
High ratio of READ_WRITE to READ_ONLY	Poor	Great
Memory consumption	Low	Higher
Processing overhead	Low	High
Concurrency/multi-core utilization	Depends on ratio of READ_ONLY to READ_WRITE	Great



For any shared resource, you must address concurrency control. The level of effort you put into this depends on the operational characteristics of the system. If there are few concurrent tasks that have a heavy preponderance of READ_ONLY to READ_WRITE operations, then a simplistic pessimistic approach will be the easiest to code and validate. As you move down the spectrum toward more concurrent tasks with a greater ratio of READ_WRITE to READ_ONLY operations, need to deal with slower (than DRAM) media, and/or the resource-holding time increases, then the investment in an optimistic approach begins to be justified.

About the Author

Steve Graves co-founded McObject in 2001. As the company's president and CEO, he has both spearheaded McObject's growth and helped the company attain its goal of providing embedded database technology that makes embedded systems smarter, more reliable and more cost-effective to develop and maintain. Prior to McObject, Graves was president and chairman of Centura Solutions Corporation, and vice president of worldwide consulting for Centura Software Corporation (NASDAQ: CNTR); he also served as president and chief operating officer of Raima Corporation. Graves is a member of the advisory board for the University of Washington's certificate program in Embedded and Real Time Systems Programming. For Steve's updates on McObject, embedded software and the business of technology, follow him on Twitter.

Embedded database professional with 25+ years of experience in roles ranging from programmer/consultant to VP of engineering, to president/COO and currently entrepeneur of McObject LLC, an in-memory embedded database vendor. Particularly interested in contacts with other embedded systems middleware vendors for exploration of potential alliances.

More from Steve

Subscribe to None

Submit Comment