Peter Bailis :: Highly Available, Seldom Consistent

Data management, distributed systems, and beyond

Linearizability versus Serializability

24 Sep 2014

Linearizability and serializability are both important properties about interleavings of operations in databases and distributed systems, and it's easy to get them confused. This post gives a short, simple, and hopefully practical overview of the differences between the two.

Linearizability: single-operation, single-object, real-time order

Linearizability is a guarantee about single operations on single objects. It provides a real-time (i.e., wall-clock) guarantee on the behavior of a set of single operations (often reads and writes) on a single object (e.g., distributed register or data item).

In plain English, under linearizability, writes should appear to be instantaneous. Imprecisely, once a write completes, all later reads (where "later" is defined by wall-clock start time) should return the value of that write or the value of a later write. Once a read returns a particular value, all later reads should return that value or the value of a later write.

Linearizability for read and write operations is synonymous with the term "atomic consistency" and is the "C," or "consistency," in Gilbert and Lynch's proof of the CAP Theorem. We say linearizability is *composable* (or "local") because, if operations on each object in a system are linearizable, then all operations in the system are linearizable.

Serializability: multi-operation, multi-object, arbitrary total order

Serializability is a guarantee about transactions, or groups of one or more operations over one or more objects. It guarantees that the execution of a set of transactions (usually containing read and write operations) over multiple items is equivalent to *some* serial execution (total ordering) of the transactions.

Serializability is the traditional "I," or isolation, in ACID. If users' transactions each preserve application correctness ("C," or consistency, in ACID), a serializable execution also preserves correctness. Therefore, serializability is a mechanism for guaranteeing database correctness.[1]

Unlike linearizability, serializability does not—by itself—impose any real-time constraints on the ordering of transactions. Serializability is also not composable. Serializability does not imply any kind of deterministic order—it simply requires that *some* equivalent serial execution exists.

Strict Serializability: Why don't we have both?

Combining serializability and linearizability yields *strict serializability*: transaction behavior is equivalent to some serial execution, and the serial order corresponds to real time. For example, say I begin and commit transaction T1, which writes to item *x*, and you later begin and commit transaction T2, which reads from *x*. A database providing strict serializability for these transactions will place T1 before T2 in the serial ordering, and T2 will read T1's write. A database providing serializability (but not strict serializability) could order T2 before T1.[2]

As Herlihy and Wing note, "linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object."

Coordination costs and real-world deployments

Neither linearizability nor serializability is achievable without coordination. That is we can't provide either guarantee with availability (i.e., CAP "AP") under an asynchronous network.[3]

In practice, your database is unlikely to provide serializability, and your multi-core processor is unlikely to provide linearizability—at least by default. As the above theory hints, achieving these properties requires a lot of expensive coordination. So, instead, real systems often use cheaper-to-implement and often harder-to-understand models. This trade-off between efficiency and programmability represents a fascinating and challenging design space.

A note on terminology, and more reading

One of the reasons these definitions are so confusing is that linearizability hails from the distributed systems and concurrent programming communities, and serializability comes from the database community. Today, almost everyone uses *both* distributed systems and databases, which often leads to overloaded terminology (e.g., "consistency," "atomicity").

There are many more precise treatments of these concepts. I like this book, but there is plenty of free, concise, and (often) accurate material on the internet, such as these notes.

Notes

1. But it's not the only mechanism!

   Granted, serializability is (more or less) the most *general* means of maintaining database correctness. In what's becoming one of my favorite "underground" (i.e., relatively poorly-cited) references, H.T. Kung and Christos Papadimitriou dropped a paper in SIGMOD 1979 on "An Optimality Theory of Concurrency Control for Databases." In it, they essentially show that, if all you have are transactions' syntactic modifications to database state (e.g., read and write) and *no* information about application logic, serializability is, in some sense, "optimal": in effect, a schedule that is not serializable might modify the database state in a way that produces inconsistency for some (arbitrary) notion of correctness that is not known to the database.

   However, if *do* you know more about your user's notions of correctness (say, you *are* the user!), you can often do a lot more in terms of concurrency control and can circumvent many of the fundamental overheads imposed by serializability. Recognizing when you don't need serializability (and subsequently exploiting this fact) is the best way I know to "beat CAP." ↩

2. Note that some implementations of serializability (such as two-phase locking with long write locks and long read locks) actually provide strict serializability. As Herlihy and Wing point out, other implementations (such as some MVCC implementations) may not.

   So, why didn't the early papers that defined serializability call attention to this real-time ordering? In some sense, real time doesn't really matter: all serializable schedules are equivalent in terms of their power to preserve database correctness! However, there are some weird edge cases: for example, returning NULL in response to every read-only transaction is serializable (provided we start with an empty database) but rather unhelpful.

   One tantalizingly plausible theory for this omission is that, back in the 1970s when serializability theory was being invented, everyone was running on single-site systems anyway, so linearizability essentially "came for free." However, I believe this theory is unlikely: for example, database pioneer Phil Bernstein was already looking at distributed transaction execution in his SDD-1 system as early as 1977 (and there are older references yet). Even in this early work, Bernstein (and company) are careful to stress that "there may in fact be *several* such equivalent serial orderings" [emphasis theirs]. To further put this theory to rest, Papadimitriou makes clear in his seminal 1979 JACM article that he's familiar with problems inherent in a distributed setting. (If you ever want to be blown away by the literature, look at how much of the foundational work on concurrency control was done by the early 1980s.) ↩

3. For distributed systems nerds: achieving linearizability for reads and writes is, in a formal sense, "easier" to achieve than serializability. This is probably deserving of another post (encouragement appreciated!), but here's some intuition: terminating atomic register read/write operations are achievable in a fail-stop model. Yet atomic commitment—which is needed to execute multi-site serializable transactions (think: AC is to 2PC as consensus is to Paxos)—is not: the FLP result says consensus is unachievable in a fail-stop model (hence *with One Faulty Process*), and (non-blocking) atomic commitment is "harder" than consensus (see also). Also, keep in mind that linearizability for read-modify-write is harder than linearizable read/write. (linearizable read/write 《 consensus 《 atomic commitment) ↩

You can follow me on Twitter here.

7 Comments **Highly Available, Seldom Consistent** 1 **Login**

Recommend 6 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Samy Al Bahra • 5 years ago • edited

Linearizability doesn't provide any real-time guarantees. What do you mean by "real-time" here?

1 • Reply • Share ›

Peter Bailis Samy Al Bahra • 5 years ago • edited

I'm not sure I follow -- can you elaborate? Ostensibly, one could imagine a history that is not consistent with real-time ordering, but even Herlihy and Wing concede that " [i]nformally, <_H captures the 'real-time' precedence ordering of operations in H."

edit: If you're referring to "real-time" in the sense of scheduling deadlines, at least in the distributed case, I'd agree that linearizability is hard to guarantee given a hard real-time deadline. However, in a single-node context, linearizability is still considerably friendlier towards deadline-based approaches than more coordination-intensive semantics like serializability (see the bit in the post above).

• Reply • Share ›

Samy Bahra Peter Bailis • 5 years ago

That clears it up, thanks. A "real-time guarantee" means something else today and in the literature I have come across. Perhaps different terminology should be used or a footnote to clarify this point?

Unfortunately, in a single-node context you are still at the will of the underlying scheduler and hardware depending on the progress guarantee of your linearized operation. In many cases, you may find serialized forms of synchronization make it easier to provide actual deadline guarantees (where linearized counterparts