THE BROADER SPECTRUM OF SEMANTICS

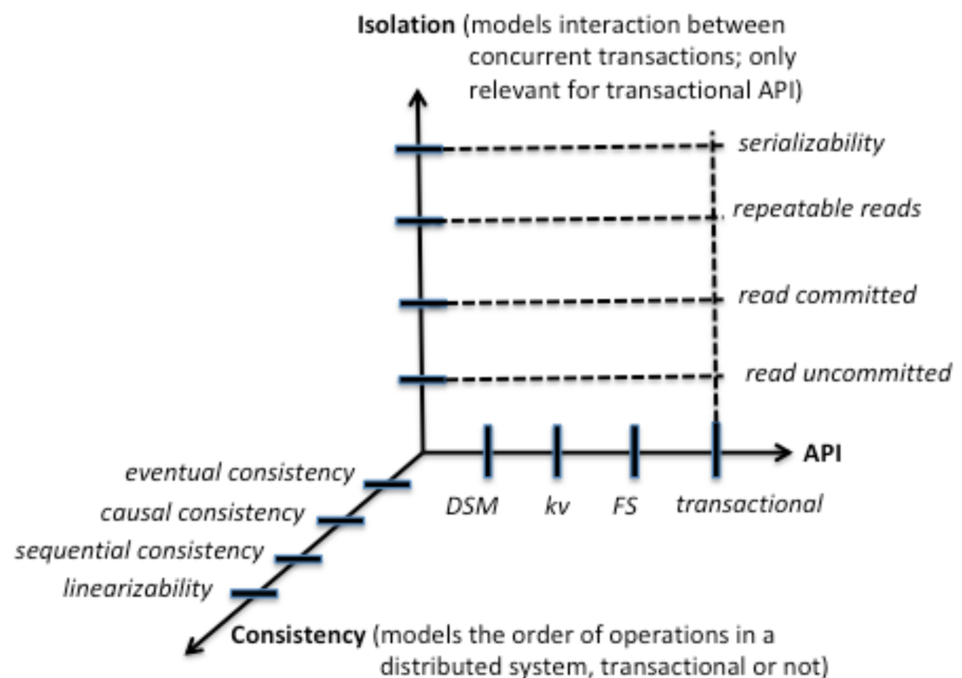
Thus far, we have looked at mechanisms to achieve some of *the strongest possible semantics* for several properties of storage systems: transactional interface, atomicity, serializable isolation, externally consistent ordering of transactions. We've looked at some methods to achieve these in both local and distributed settings. We've also shown how these methods are being used in a real-world system -- Google's Spanner. The reason we started with the strongest possible semantics is that they are the most intuitive from a human/programmer perspective: they approximate the best the behavior of a non-distributed system, non-concurrent system, which is the easiest for our serial minds to think about. But the mechanisms needed to achieve these strong semantics and APIs *are expensive*. To quote the Spanner paper again:

> **"We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions."** (https://www.usenix.org/node/170855)

(Here, what I believe they mean by "transactions" is both the interface and the full set of strong semantics they enforce for them: ACID, external consistency).

Still, there are situations when it may make more sense to use a storage system that provides weaker *but still well defined* semantics and APIs instead of optimizing one's code on top of a storage system that has strong semantic but it's just too expensive.

The figure to the right maps a broader view of the spectrum of semantics and APIs. Spanner would be a point in this 3-D graph that joins serializability, linearizability, and transactional API. Not all intersections of various semantics form meaningful design points. In particular, isolation is a property that is relevant only to transactional APIs.



In this lecture, we give a broader overview of the spectrum of well-defined semantics for the *isolation and consistency properties*. As part of consistency, we'll look at non-transactional APIs, too. This particular document covers isolation semantics.

I. ISOLATION SEMANTICS

The isolation semantics of a system with transactional API specifies the *guarantees that the system gives with respect to how* ***multi-operation transactions*** *are allowed to interact under concurrency*. The golden standard to "imitate" is the single-threaded, non-concurrent, non-replicated system, where the system executes transactions serially. All transactions run unaffected by other transactions (so no interaction).

There are multiple ways of enforcing various isolation semantics. Some use locks, others don't. The former are called lock-based concurrency control mechanisms, the latter are called optimistic concurrency control mechanisms. We've talked most about the former and a bit about the latter (as part of Spanner's multi-version transactional system that supports lockless read-only queries). When defining various levels of semantics, it's easiest to think about the lock-based mechanisms that can be used to yield each level, as they make very clear why/how the semantic gets weaker but more efficient to implement at every new level of the semantic:

Serializability is equivalent to the golden standard of scheduling concurrent transactions sequentially one after another with no overlapping transactions. It imposes no restriction on the order of how transactions are actually committed. Full 2PL can enforce this sematic, but it's expensive: it requires grabbing row-level locks for read/written rows, plus range locks for any range-based read query. It requires maintaining these locks until the transaction completes. It can therefore be very expensive, esp. if transactions contain range queries: during the course of the transaction, no one can insert/update any row within that range. As we discussed in class, for certain types of queries, range locks can degrade into whole-table locks!

The following isolation semantics, which are usually offered as options by commercial database implementations, can be thought of as progressively getting rid of some of the locking that's needed to achieve serializability, therefore improving performance but at the cost of permitting progressively more unintuitive forms of interaction between concurrent transactions. So, they move further and further away from the golden standard of a single-threaded/single-process/single-machine system.

Repeatable Reads: This semantic is what you get using a 2PL mechanism that *does not grab any range locks*: i.e., you only grab row-level locks for any rows that are read or written and retain those till the end of the transaction. That removes the most expensive aspect of a full 2PL, hence the performance improvement. The negative effect is that it permits what are called *phantom reads*. If you do have a range query in your transaction, then the set of rows relevant for that query may change through the course of your transaction (e.g., another concurrent transaction may insert a new row in the range relevant to your query, hence the set of rows your query would return depends on whether that transaction's insert operation is executed before or after your range query). On the other hand, the *values* for any rows you have read as part of your transaction will not change through the course of your transation, because you still grab locks for all individual rows you read/write. This is why this semantic is called *repeatable reads*: the values of any rows being read as part of the transaction do not change depending on when you read them as part of your transaction and what other transactions are running in the system.

Read Committed: This semantic is the same as what you get by grabbing long-term locks only for written rows; for any read rows, you grab short-term row-level locks, i.e., you grab them but retain them only while you execute that specific read query and not till the end of the transaction. You grab no range locks. The good parts of this semantic are that (1) you get rid of long-term locks for read rows, which further increases performance and (2) it doesn't allow you to read partially written versions of rows, or rows that contain effects of uncommitted transactions. (2) justifies why the semantic is called *read committed*: you can only read effects of committed transactions. However, this semantic can permit both *phantom reads* (see above) and what are called *non-repeatable reads*. What this means is that not only the *sets* of rows you read but also the *values* of these rows may change depending on when you read them as part of your transaction and what other transactions are running in the system.

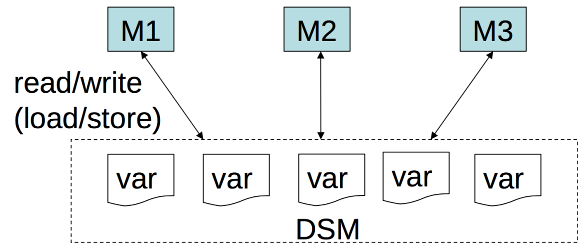
Read Uncommitted: This semantic is the same as what you get by grabbing long-term locks for written rows, but no row-level or range locks for read queries AT ALL. This means that while concurrent writers to the same row cannot trip each other and result in rows that contain effects mixed from multiple transactions, the semantic of reads is undefined. This semantic can permit what is called *dirty reads*, i.e., reads of transient effects of uncommitted transactions. This means that you can read the effect of a transaction that will eventually end up being rolled back. Hence the name *read uncommitted*.

Most commercial databases offer default of either **Repeatable Reads** or **Read Committed**. Read Uncommitted is generally perceived as too low semantic for a transactional database while serializable isolation is often considered too expensive (not in Spanner though!). The programmer thus needs to explicitly choose the serializable level of isolation if that semantic is needed and can be afforded.

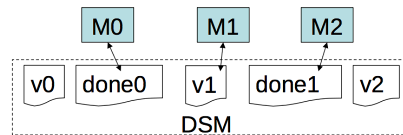
II. CONSISTENCY SEMANTICS

We've talked about Spanner, a system that implements a very strong consistency semantic for serializable transactions: *external consistency*, which when you apply to a non-transactional API is known as *linearizability*. In a linearizable system, any execution history of the system is equivalent to a serial ordering of all operations in which (1) program order is preserved and (2) if a transaction T2 begins after another transaction T1 commits in real time, then T1 will appear before T2 in that serial ordering. To achieve this semantic for all operations, including for range queries, Spanner relies on TrueTime to timestamp transactions and wait out uncertainty to ensure the preceding rule (2) holds. But what if you don't have TrueTime, or the atomic/GPS clocks it depends on in your datacenter? You can achieve various degrees of consistency without having tightly synchronized clocks, but instead using logical clocks and version vectors. In this lecture, we discuss various consistency models and methods for implementing them, to give you a sense of the range of possibilities. The main takeaway from this lecture is that higher degrees of consistency are more expensive to achieve but they are easier to think about when programming on top of them.

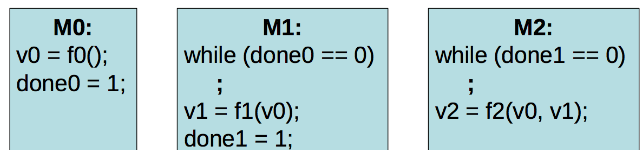
A. An Example: Distributed Shared Memory



It's hard to think about consistency models in the context of multi-operation transactions, so in this lecture we'll think about individual operations on individual registers/variables residing in a distributed shared memory system (DSM). See figure to the right. M1, M2, and M3 are clients (application-level nodes) accessing variables in the DSM. The DSM, which can be implemented either as a library or a separate service, performs all synchronization needed to give the illusion of a distributed memory space accessible to all machines. Multiple implementations are possible for the DSM, and different implementations may offer different properties along the performance, scalability, consistency semantics, and programmability axes. We'll look at some implementations that differ in the consistency semantics they provide, and the implications for programmability, performance and scalability.

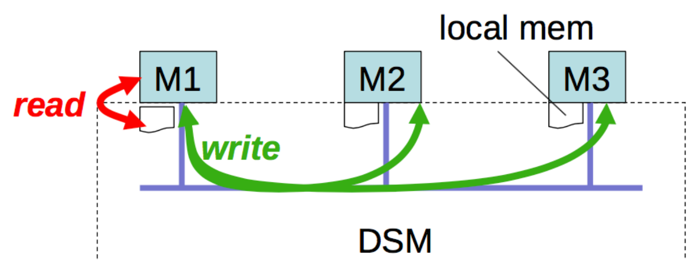


To drive the discussion and comparison of consistency semantics of different implementations, imagine the following application implemented by M0, M1, and M2 (figure to the right). The intuitive intent of this application is that M1 should execute f1() with the result from M0, and M2 should execute f2() with results from M0 and M1.



We'll look at various DSM implementations and evaluate whether they can support this application (as written) or not.

B. A Naïve DSM Implementation



We'll start with a very naïve implementation of DSM, which offers no clear semantic. Each machine has a local copy of all of memory.

Operations:

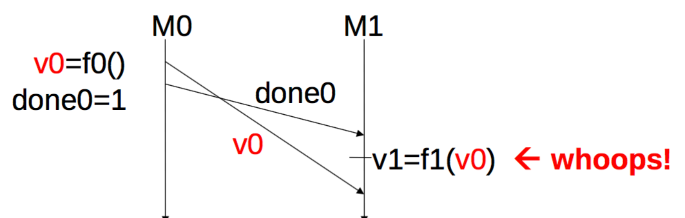
- **Read**: from local memory
- **Write**: send update msg to each host, but **don't wait** until the update is acknowledged before write returns.

Properties:

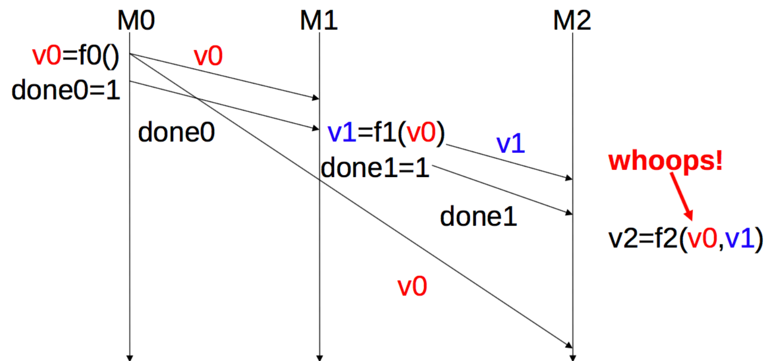
- Fast: never waits for communication.
- But: does this DSM implementation support our preceding application?
  - Answer: no, two problems, listed below.

<u>Consistency problem 1:</u>

M0's v0=… and done0=… may be interchanged by the network, leaving v0 unset but done0=1. This is an example of breaking program order.



Consistency problem 2:
M2 sees M1's writes before M0's writes. I.e., M2 and M1 disagree on the ordering of M0 and M1 writes. This is an example of lack of a global order for writes.



So, the naïve DSM is fast but has unexpected behavior.

- Maybe DSM isn't "correct"
- Or maybe we should have never expected the example application to work in the first place. I.e., maybe we need to fix the app, not the DSM...
- How would you "fix" the app to achieve the desired semantic with the given DSM? (Challenging right?)

C. Consistency Models

A memory or storage system's consistency model consists of a concise and precise set of rules that govern the possible behaviors of operations (e.g., in the case of our simple DSM, what values read and write operations return) in the face of failures and reorderings. This set of rules constitute the DSM's contract with the programmer: it's what the programmer uses to engineer his/her application. All consistency models make certain assumptions, particularly related to failures, but those need to be stated concisely and precisely, too.

While the lack of an easily expressible consistency semantic is problematic, there's generally no right or wrong consistency model. The tradeoff is always between ease of programming and efficiency + availability. But generally, when designing a new system, it's useful to consider choosing one of the several standard consistency semantics that exist and supporting that. The reason is that these are well understood and so programmability might actually increase on top of your system if you offer a familiar semantic.

Here we look at four standard consistency models in addition to linearizability, which we've already introduced in Spanner:

- Strict consistency
- Sequential consistency
- Causal consistency
- Eventual consistency

These go from stronger, easier to program models at the top to more efficient, available and scalable at the bottom. Variations boil down to: (1) *the allowable staleness of reads* and (2) *the ordering of writes across the DSM nodes*. The consistency semantics, briefly defined below, are best described with examples, which you can find in these slides: https://columbia.github.io/ds1-class/lectures/09-consistency-models-slides.pdf.

Brief definitions of each model for quick reference:

- Strict consistency: Any execution is the same as if all read/write ops were executed in order of wall-clock time at which they were issued.
- Linearizability: Any execution is the same as if all read/write ops were executed in some global order s.t. any read returns the value of the most recent write operation completed at that location.
- Sequential consistency: Any execution is the same as if all read/write ops were executed in some global order, and the ops of each client process appear in the order specified by its program.
- Causal consistency: Any execution is the same as if all causally-related read/write ops were executed in an order that reflects their causality. All *concurrent* operations may be seen in different orders by different clients. So there doesn't need to exist a global order.
- Eventual consistency: Allow stale reads, but ensure that reads will eventually reflect previously written values.

It's worth reflecting on which of these consistency models can support the application given as an example in Section II, and specifically whether they can remove the two problems we discussed. Hints at the answers (please reflect on why / a proof sketch): strict consistency – yes; linearizability – yes, sequential consistency – yes; causal consistency – yes; eventual consistency – not necessarily.

D. Implementation Examples and Tradeoffs

Sequential consistency:

- Example 1: Primary-based architecture: fully replicated DBs, one node is assigned as the primary, updates go to the primary, which assigns them an order; block until they are acknowledged; reads go to the primary. (Primaries are elected through consensus.)
- Example 2: Chain replication (OSDI'04): fully replicated DBs, replicas are organized in a chain; updates go to the head of the chain, where they are assigned a unique ID; they traverse the chain and are finally ack'ed by the tail of the chain; reads go to the tail of the chain. Please read the paper here: https://www.cs.cornell.edu/home/rvr/papers/OSDI04.pdf.
- Example 3: Ivy DSM (1980s): https://columbia.github.io/ds1-class/lectures/09-consistency-models-ivy.pdf
- In general, implementations require synchronous update propagation, which is expensive, esp. over the WAN and can be affected by cross-DC partitions for example.

Causal consistency:

- You can implement it with lazier replication.
- Example: The COPS system, presented at SOSP 2011, arranges replicas into clusters, where each cluster resides in a single datacenter. Writes within a cluster are strictly ordered, resulting in sequential consistency (actually, linearizability) within a cluster. Causal ordering between Writes is tracked by the client library in the form of dependencies. Writes are propagated between clusters lazily. When writes are propagated across clusters, they carry their dependencies with them. A cluster only commits a Write (i.e., exposes its effects) when all of its dependencies have already been committed in that cluster. This ensures that replicas within each cluster remain causally consistent. Reads are allowed to access only a single cluster. This ensures that readers see a causally consistent system. Details about the COPS design are available in the SOSP 2011 talk: https://www.youtube.com/watch?v=jh9P1moDpAc.
- Lazier replication, along with the assumption that a reader goes to one single cluster, allows causal consistency to implement lazy replication across clusters and hence withstand cross-DC

partitions. This means that causal consistency can give lower latency and higher availability compared to sequential consistency. But, remember its odd behaviors, which make programming pretty challenging on this model.

Eventual consistency:

- Even lazier replication. Can support offline operation (i.e., no assumptions about the type of partitions), however it doesn't give any guarantees about the eventual ordering of operations.
- Example 1: think git. Example 2: Amazon's Dynamo distributed key/value store: https://dl.acm.org/citation.cfm?id=1294281. Example 3: discuss file synchronizer (from slides).

E. Reflection

The preceding discussion about consistency levels vs. availability with network partitions can be formalized. The Consistency-Availability-Partitions (CAP) Theorem, initially conjectured by Eric Brewer and then proven by Gilbert and Lynch, states that in a system where network partitions can happen, one has to choose between linearizability and availability. One cannot get both – provably so. There are additional theorems that prove similar tradeoffs for sequential consistency and other consistency models stronger than causal consistency. Causal consistency appears to be the strongest model to permit high availability in the face of certain network partitions. The COPS paper (and their SOSP talk) discuss this in more detail.

Acknowledgements

The slides linked from these notes contain portions adapted from NYU's distributed systems course, Jinyang Li's edition: (http://www.news.cs.nyu.edu/~jinyang/fa10/notes/ds-eventual.ppt). The notes include excerpts copied or inspired from: Doug Terry's blog post on causal consistency (https://littlemindslargeclouds.wordpress.com/2014/05/27/implementing-causal-consistency) and Kyle Kingsbury's blog post on consistency models (https://aphyr.com/posts/313-strong-consistency-models).