Causal Consistency and Read and Write Concerns

With MongoDB's causally consistent client sessions, different combinations of read and write concerns provide different causal consistency guarantees. When causal consistency is defined to imply durability, then the following table lists the specific guarantees provided by the various combinations:

Read Concern	Write Concern	Read own writes	Monotonic reads	Monotonic writes	Writes follow reads
`"majority"`	`"majority"`				
`"majority"`	`{ w: 1 }`				
`"local"`	`{ w: 1 }`				
`"local"`	`"majority"`				

If causal consistency implies durability, then, as seen from the table, only read operations with `"majority"` read concern and write operations with `"majority"` write concern can guarantee all four causal consistency guarantees. That is, causally consistent client sessions can only guarantee causal consistency for:

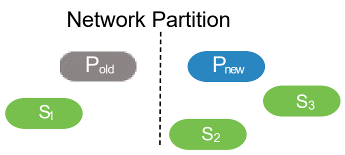
- Read operations with `"majority"` read concern; i.e. the read operations that return data that has been acknowledged by a majority of the replica set members and is durable.
- Write operations with `"majority"` write concern; i.e. the write operations that request acknowledgement that the operation has been applied to a majority of the replica set's voting members.

If causal consistency does not imply durability (i.e. writes may be rolled back), then write operations with `{ w: 1 }` write concern can also provide causal consistency.

NOTE:
The other combinations of read and write concerns may also satisfy all four causal consistency guarantees in some situations, but not necessarily in all situations.

The read concern `"majority"` and write concern `"majority"` ensure that the four causal consistency guarantees hold even in circumstances (such as with a network partition) where two members in a replica set *transiently* believe that they are the primary. And while both primaries can complete writes with `{ w: 1 }` write concern, only one primary will be able to complete writes with `"majority"` write concern.

For example, consider a situation where a network partition divides a five member replica set:



WITH THE ABOVE PARTITION:

- Writes with `"majority"` write concern can complete on P_{new} but cannot complete on P_{old}.
- Writes with `{ w: 1 }` write concern can complete on either P_{old} or P_{new}. However, the writes to P_{old} (as well as the writes replicated to S_1) roll back once these members regain communication with the rest

of the replica set.

- After a successful write with `"majority"` write concern on P_{new}, causally consistent reads with `"majority"` read concern can observe the write on P_{new}, S_2,and S_3. The reads can also observe the write on P_{old} and S_1 once they can communicate with the rest of the replica set and sync from the other members of the replica set. Any writes made to P_{old} and/or replicated to S_1 during the partition are rolled back.

Scenarios

To illustrate the read and write concern requirements, the following scenarios have a client issue a sequence of operations with various combination of read and write concerns to the replica set:

- Read Concern "majority" and Write concern "majority"
- Read Concern "majority" and Write concern {w: 1}
- Read Concern "local" and Write concern "majority"
- Read Concern "local" and Write concern {w: 1}

Read Concern `"majority"` and Write concern `"majority"`

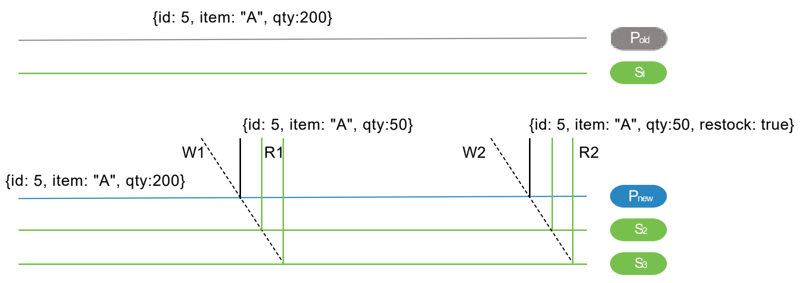
The use of read concern `"majority"` and write concern `"majority"` in a causally consistent session provides the following causal consistency guarantees:

Read own writes   Monotonic reads   Monotonic writes   Writes follow reads

Scenario 1 (Read Concern “majority” and Write Concern “majority”)

During the transient period with two primaries, because only P_{new} can fulfill writes with `{ w: "majority" }` write concern, a client session can issue the following sequence of operations successfully:

Sequence	Example
1. $Write_1$ with write concern `"majority"` to P_{new}	For item A, update `qty` to `50`.
2. $Read_1$ with read concern `"majority"` to S_2	Read item A.
3. $Write_2$ with write concern `"majority"` to P_{new}	For items with `qty` less than `50`, update `reorder` to `true`.
4. $Read_2$ with read concern `"majority"` to S_3	Read item A.


{id: 5, item: "A", qty:200}
P_old
S_1
{id: 5, item: "A", qty:50}
{id: 5, item: "A", qty:50, restock: true}
W1
R1
W2
R2
{id: 5, item: "A", qty:200}
P_new
S_2
S_3

Read own writes	$Read_1$ reads data from S_2 that reflects a state after $Write_1$. $Read_2$ reads data from S_1 that reflects a state after $Write1_1$ followed by $Write_2$.
Monotonic reads	$Read_2$ reads data from S_3 that reflects a state after $Read_1$.
Monotonic writes	$Write_2$ updates data on P_{new} that reflects a state after $Write_1$.
Writes follow reads	$Write_2$ updates data on P_{new} that reflects a state of the data after $Read_1$ (i.e. an earlier state reflects the data read by $Read_1$).

Scenario 2 (Read Concern “majority” and Write Concern “majority”)

Consider an alternative sequence where Read$_1$ with read concern "majority" routes to S$_1$:

Sequence	Example
1. Write$_1$ with write concern "majority" to P$_{new}$	For item A, update qty to 50.
2. Read$_1$ with read concern "majority" to S$_1$	Read item A.
3. Write$_2$ with write concern "majority" to P$_{new}$	For items with qty less than 50, update reorder to true.
4. Read$_2$ with with read concern "majority" to S$_3$	Read item A.

In this sequence, Read$_1$ cannot return until the majority commit point has advanced on P$_{old}$. This cannot occur until P$_{old}$ and S$_1$ can communicate with the rest of the replica set; at which time, P$_{old}$ has stepped down (if not already), and the two members sync (including Write$_1$) from the other members of the replica set.

Read own writes	Read$_1$ reflects a state of data after Write1$_1$, albeit after the network partition has healed and the member has sync'ed from the other members of the replica set. Read$_2$ reads data from S$_3$ that reflects a state after Write1$_1$ followed by Write$_2$.
Monotonic reads	Read$_2$ reads data from S$_3$ that reflects a state after Read$_1$ (i.e. an earlier state is reflected in the data read by Read$_1$).
Monotonic writes	Write$_2$ updates data on P$_{new}$ that reflects a state after Write$_1$.
Writes follow reads	Write$_2$ updates data on P$_{new}$ that reflects a state of the data after Read$_1$ (i.e. an earlier state reflects the data read by Read$_1$).

Read Concern "majority" and Write concern {w: 1}

The use of read concern "majority" and write concern { w: 1 } in a causally consistent session provides the following causal consistency guarantees *if causal consistency implies durability*:

Read own writes Monotonic reads Monotonic writes Writes follow reads

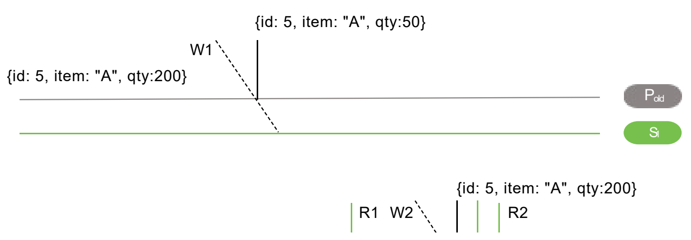
If causal consistency does not imply durability:

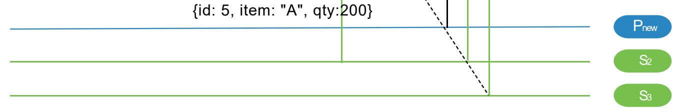
Read own writes Monotonic reads Monotonic writes Writes follow reads

Scenario 3 (Read Concern "majority" and Write Concern {w: 1})

During the transient period with two primaries, because both P$_{old}$ and P$_{new}$ can fulfill writes with { w: 1 } write concern, a client session could issue the following sequence of operations successfully but not be causally consistent **if causal consistency implies durability**:

Sequence	Example
1. Write$_1$ with write concern { w: 1 } to P$_{old}$	For item A, update qty to 50.
2. Read$_1$ with read concern "majority" to S$_2$	Read item A.
3. Write$_2$ with write concern { w: 1 } to P$_{new}$	For items with qty less than 50, update reorder to true.
4. Read$_2$ with with read concern "majority" to S$_3$	Read item A.


{id: 5, item: "A", qty:50}
W1
{id: 5, item: "A", qty:200}
P$_{old}$
S$_1$
{id: 5, item: "A", qty:200}
R1
W2
R2



In this sequence,

- Read$_1$ cannot return until the majority commit point has advanced on P_{new} past the time of Write$_1$.
- Read$_2$ cannot return until the majority commit point has advanced on P_{new} past the time of Write$_2$.
- Write$_1$ will roll back when the network partition is healed.

➢ *If causal consistency implies durability*

Read own writes	Read$_1$ reads data from S_2 that does not reflect a state after Write$_1$.
Monotonic reads	Read$_2$ reads data from S_3 that reflects a state after Read$_1$ (i.e. an earlier state is reflected in the data read by Read$_1$).
Monotonic writes	Write$_2$ updates data on P_{new} that does not reflect a state after Write$_1$.
Writes follow reads	Write$_2$ updates data on P_{new} that reflects a state after Read$_1$ (i.e. an earlier state reflects the data read by Read$_1$).

➢ *If causal consistency does not imply durability*

Read own writes	Read$_1$ reads data from S_2 returns data that reflects a state equivalent to Write$_1$ followed by rollback of Write$_1$.
Monotonic reads	Read$_2$ reads data from S_3 that reflects a state after Read$_1$ (i.e. an earlier state is reflected in the data read by Read$_1$).
Monotonic writes	Write$_2$ updates data on P_{new} that is equivalent to after Write$_1$ followed by rollback of Write$_1$.
Writes follow reads	Write$_2$ updates data on P_{new} that reflects a state after Read$_1$ (i.e. whose earlier state reflects the data read by Read$_1$).

Scenario 4 (Read Concern “majority” and Write Concern `{w: 1}`)

Consider an alternative sequence where Read$_1$ with read concern `"majority"` routes to S_1:

Sequence	Example
Write$_1$ with write concern `{ w: 1 }` to P_{old}	For item `A`, update `qty` to `50`.
Read$_1$ with read concern `"majority"` to S_1	Read item `A`.
Write$_2$ with write concern `{ w: 1 }` to P_{new}	For items with `qty` less than `50`, update `reorder` to `true`.
Read$_2$ with with read concern `"majority"` to S_3	Read item `A`.

In this sequence:

- Read$_1$ cannot return until the majority commit point has advanced on S_1. This cannot occur until P_{old} and S_1 can communicate with the rest of the replica set. At which time, P_{old} has stepped down (if not already), Write$_1$ is rolled back from P_{old} and S_1, and the two members sync from the other members of the replica set.

➢ *If causal consistency implies durability*

Read own writes	The data read by Read$_1$ does not reflect the results of Write$_1$, which has rolled back.

Monotonic reads	$Read_2$ reads data from S_3 that reflects a state after $Read_1$ (i.e. whose earlier state reflects the data read by $Read_1$).
Monotonic writes	$Write_2$ updates data on P_{new} that does not reflect a state after $Write_1$, which had preceded $Write_2$ but has rolled back.
Writes follow reads	$Write_2$ updates data on P_{new} that reflects a state after $Read_1$ (i.e. whose earlier state reflects the data read by $Read_1$).

➤ *If causal consistency does not imply durability*

Read own writes	$Read_1$ returns data that reflects the final result of $Write_1$ since $Write_1$ ultimately rolls back.
Monotonic reads	$Read_2$ reads data from S_3 that reflects a state after $Read_1$ (i.e. an earlier state reflects the data read by $Read_1$).
Monotonic writes	$Write_2$ updates data on P_{new} that is equivalent to after $Write_1$ followed by rollback of $Write_1$.
Writes follow reads	$Write_2$ updates data on P_{new} that reflects a state after $Read_1$ (i.e. an earlier state reflects the data read by $Read_1$).

Read Concern `"local"` and Write concern `{w: 1}`

The use of read concern `"local"` and write concern `{ w: 1 }` in a causally consistent session cannot guarantee causal consistency.

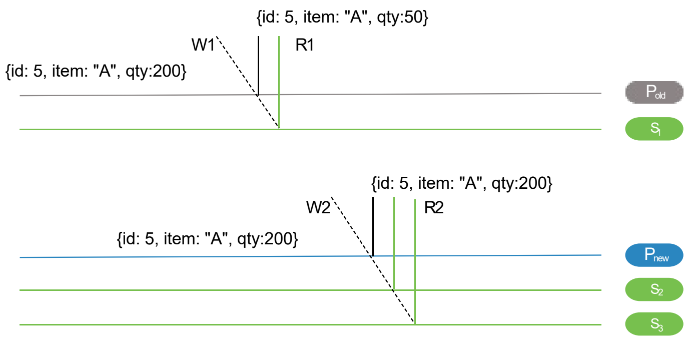
Read own writes Monotonic reads Monotonic writes Writes follow reads

This combination may satisfy all four causal consistency guarantees in some situations, but not necessarily in all situations.

Scenario 5 (Read Concern “local” and Write Concern `{w: 1}`)

During this transient period, because both P_{old} and P_{new} can fulfill writes with `{ w: 1 }` write concern, a client session could issue the following sequence of operations successfully but not be causally consistent:

Sequence	Example
1. $Write_1$ with write concern `{ w: 1 }` to P_{old}	For item `A`, update `qty` to `50`.
2. $Read_1$ with read concern `"local"` to S_1	Read item `A`.
3. $Write_2$ with write concern `{ w: 1 }` to P_{new}	For items with `qty` less than `50`, update `reorder` to `true`.
4. $Read_2$ with read concern `"local"` to S_3	Read item `A`.


{id: 5, item: "A", qty:50}
W1
R1
{id: 5, item: "A", qty:200}
P old
S1
{id: 5, item: "A", qty:200}
W2
R2
{id: 5, item: "A", qty:200}
P new
S2
S3

Read own writes	$Read_2$ reads data from S_3 that only reflects a state after $Write_2$ and not $Write_1$ followed by $Write_2$.
Monotonic reads	$Read_2$ reads data from S_3 that does not reflect a state after $Read_1$ (i.e. an earlier state does not reflect the data read by $Read_1$).

Monotonic writes	Write$_2$ updates data on P$_{new}$ that does not reflect a state after Write$_1$.
Write follow read	Write$_2$ updates data on P$_{new}$ that does not reflect a state after Read$_1$ (i.e. an earlier state does not reflect the data read by Read$_1$).

Read Concern **`"local"`** and Write concern **`"majority"`**

The use of read concern `"local"` and write concern `"majority"` in a causally consistent session provides the following causal consistency guarantees:

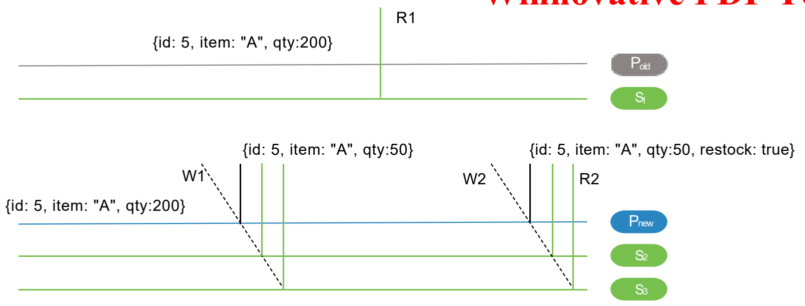
Read own writes	Monotonic reads	Monotonic writes	Writes follow reads

This combination may satisfy all four causal consistency guarantees in some situations, but not necessarily in all situations.

Scenario 6 (Read Concern “local” and Write Concern “majority”)

During this transient period, because only P$_{new}$ can fulfill writes with `{ w: "majority" }` write concern, a client session could issue the following sequence of operations successfully but not be causally consistent:

Sequence	Example
1. Write$_1$ with write concern `"majority"` to P$_{new}$	For item `A`, update `qty` to `50`.
2. Read$_1$ with read concern `"local"` to S$_1$	Read item `A`.
3. Write$_2$ with write concern `"majority"` to P$_{new}$	For items with `qty` less than `50`, update `reorder` to `true`.
4. Read$_2$ with read concern `"local"` to S$_3$	Read item `A`.


R1
{id: 5, item: "A", qty:200}
P_old
S_1
{id: 5, item: "A", qty:50}
{id: 5, item: "A", qty:50, restock: true}
W1
W2
R2
{id: 5, item: "A", qty:200}
P_new
S_2
S_3

Read own writes.	Read$_1$ reads data from S$_1$ that does not reflect a state after Write1$_1$.
Monotonic reads.	Read$_2$ reads data from S$_3$ that does not reflect a state after Read$_1$ (i.e. an earlier state does not reflect the data read by Read$_1$).
Monotonic writes	Write$_2$ updates data on P$_{new}$ that reflects a state after Write$_1$.
Write follow read.	Write$_2$ updates data on P$_{new}$ that does not reflect a state after Read$_1$ (i.e. an earlier state does not reflect the data read by Read$_1$).

← Read Isolation, Consistency, and Recency

Distributed Queries →