XDCR, Tombstone Purger and UPR in Couchbase 3.0

Contacts

For any question, comments and suggestions, please address to

-- Junyi Xie (junyi@couchbase.com),
-- Damien Katz (damien@couchbase.com)

Purpose of Document

In this doc we will explain issues that indexer, XDCR and other UPR clients may face when interacting with tombstone purging, which has been introduced since Couchbase Server 2.2. We will also discuss proposed solutions to address such issues. The document is part of UPR project in Couchbase 3.0. In order we will describe the problem, elaborate motivation and proposed solution in in terms of high level architecture, protocol, algorithms and high level data structures, but we will not dive into implementation details in each component because it is out of the scope of this document. To unify terminology, we will use the term "partition" to refer to "vbucket" in Couchbase Server.

UPR

Couchbase 3.0 will introduce UPR, a new replication protocol, that is useful for both intra-datacenter replication, external indexing, incremental backups, or anything that wants to be aware of changes that happen in the database. It uses state replication, update sequencing and MVCC snapshotting to allow for strong or eventual consistency during normal operations of all replication clients.

Tombstone Purger

Tomb purger is an optimization Couchbase Server will introduce in 2.2. Before 2.2, Couchbase Server storage engine would keep all deleted items on disk as "tombstone", which is required by XDCR and other clients to recognize deleted items from brand new items and thus enforce correct conflict resolution algorithm during replication. The major downside of storing the tombstones is that server may end up with unbounded storage space and significantly slow down disk operations for the server. Tombstone purger remove old deletion records after deletion has been persisted for a given amount of time.

Problem with Tombstone Purger

As of 2.2, the db purge uses a parameterized threshold to determine a persisted deletion should be purged or not. For example, assume the deletion “expiration time” threshold is 7 days, then all deletions that have been persisted for more than 7 days will be permanently purged from the storage. The major problem is deletions may have been purged before they are seen by client. In the case of XDCR, there is no guarantee that deletion will be replicated before it is eliminated without any trace by the purger. This may possibly break XDCR correctness if improperly handled. In particular,

1. Lack of justification. The confidence interval of deletion expiration time in db purger is static and predetermined by parameters. In practice, it is really hard, if not impossible, to give a good estimation of the time when the deletion can be safely purged. Different UPR clients may demonstrate different behavior in practice, and there is no way to preset a threshold of deletion expiration time that is good for each client. Underestimating the expiration time will cause XDCR to lose deletions, while overestimation, e.g., on order of days, will make storage unnecessarily suffer from increasing disk space buildup for days.

2. Lack of correctness guarantee. Even the deletion expiration time is on order of days, there is no guarantee that XDCR (and probably other clients) will not miss a deletion. This essentially means XDCR loses the guarantee that master and replica will be eventually consistent. The consequence of missing a deletion is multi-fold. It not only makes remote replica out-of-sync with source which fundamentally invalidates the backup functionality of XDCR, but it is infectious and may cause data loss on master cluster and other replicas. For example if users create a bi-directional XDCR and or a loop topology, a deleted doc which has been purged before XDCR replicate it to remote cluster, may resurrect back on master cluster.

3. Lack of practical efficiency and flexibility. In practice, for most cases XDCR is unlikely to lag behind master and accumulate mutations pending replication for more than 7 days. As far as we know at time of writing, we have never seen any customer use-cases such that the remote replica is days behind master cluster. Therefore setting deletion lifetime to 7 days is unnecessary in most use-cases, because it would just accumulate days of mutations and bloated storage space for nothing. In addition, different clients may demonstrate different characteristics when consuming mutations. It is hard to imagine that remote replicas are days behind master, but it is not uncommon that people backup their data stored in master cluster once every week. A universal threshold is simply not flexible.

4. Lack of diagnosability and manageability. To make things even worse, if clients like XDCR miss a deletion, there is no obvious way to fix it. The damage is permanent because the deletion is lost permanently. Today the best XDCR can do is to dump a warning message in logs saying that there exist at least one missed deletion, but users have no idea how many deletions are lost and which keys are missed, not to mention to restore them. If the key resurrect later, users have no way to tell if it happens. Users can always delete/restart XDCR from scratch if a deletion is missed,, but first this will be too expensive and users may not be able to afford it unless it is a small-scale and simple use-case, and second in some cases even restarting XDCR will not work if the deleted mutation has already been propagated back to source before users are aware of it, causing a permanent data loss since the deletion no longer exists anywhere in our system.

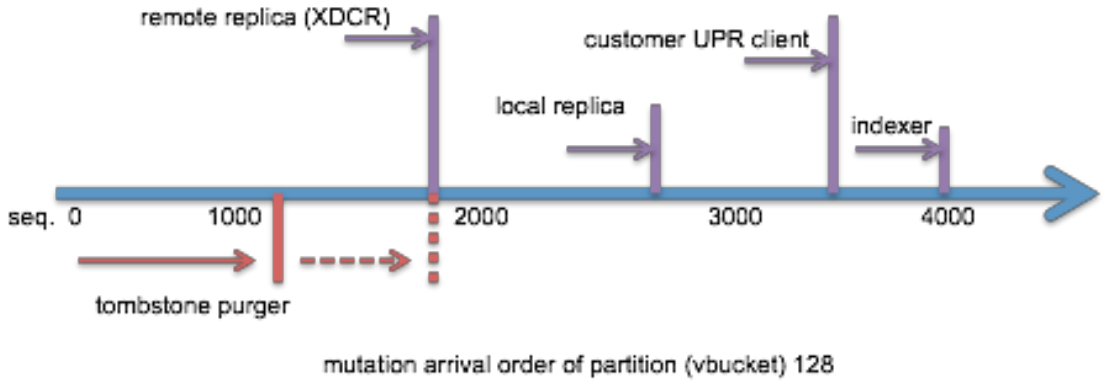
5. Other than functionality impacts, there exist other performance impacts (e.g., endlessly replicating a deletion in a loop topology) to XDCR.

In brief, the major issue of tombstone purger in 2.2 is lack of guarantee, which is we would like to address in this document.

Proposed Solution

Instead of requiring tombstone purger to blindly set threshold of deletion expiration lifetime, it is better to ask each client register its own high watermark of current progress, in terms of the partition-level sequence number, and therefore tombstone purger can be aware of the progress of each client and determine how far it can go and purge deletions without making client miss any deletions. For example, if at a time XDCR replicator has approached to the seq 123, the tombstone purger for that partition should not go beyond sequence 123. If there are multiple clients consuming mutations, each of them need to register their progress and tombstone purger will pick up the minimum seq from the "slowest" client.

The figure below shows that there are multiple UPR client consuming partition (vbucket) 128, with different rate of consumption. Assume the tombstone purger has purged all deletions up to sequence number 1000, and when it is waken up and need to determine where to advance, it will examine all registered UPR clients and figure out the minimum high watermark of all clients, e.g. XDCR is the slowest client and has replicated all mutations up to sequence 2000. Then the tombstone purger can safely purge to sequence 2000 without problem.



mutation arrival order of partition (vbucket) 128

High-level Design

High Watermark Table (HWT)

Each UPR client will need to register the high watermark to tombstone purger and we will store all high watermark in the High Watermark Table (HWT). The schema of HWT will be

HWT of partition 128 at time 2013-08-08T9:44:19

UPR client	Instance ID (UUID)	Reported High	Report Timestamp	Meta Logs

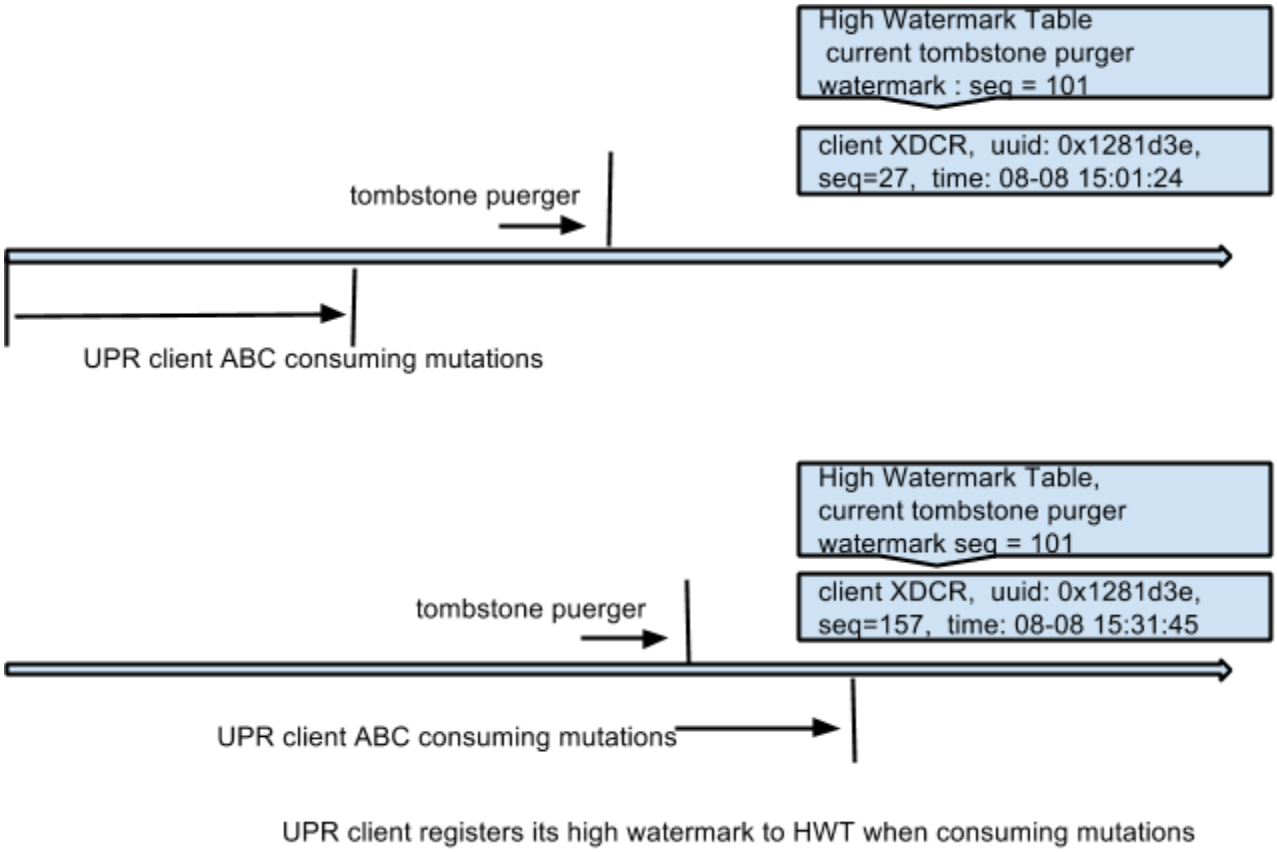
		Watermark		
XDCR	0x5c57	1008	2013-08-08T9:37:19	XDCR checkpoint log
Indexer	0x7ab1	3134	2013-08-08T9:41:21	
Rebalancer	0xe38e	2536	2013-08-08T8:12:19	checkpoint
Local replication	0x4bde	7982	2013-08-08T9:07:57	
Backup tool	0xa3ef	302	2013-08-08T3:21:45	null

Each row of HWT records UPR client, instance ID (UUID), high watermark reported from client, and timestamp of reporting, and client-specific meta logs. Each client can implement its own meta logs, for example, the meta logs for XDCR will be XDCR checkpoint logs. Whenever XDCR persist checkpoint, it records current sequence number (highwater mark) into checkpoint logs and also register it with timestamp to HWT.

Each client will determine how its high watermark is created and reported in the best way. For example, XDCR can combine checkpointing with reporting high watermark, that is, when XDCR replicator does checkpointing, it also reports the current sequence number of HWT. This will make minimum performance impact to ongoing XDCR, although the high watermark reported by XDCR may lag as long as 30 minutes. Given the deletion expiration time is now on order of days, it is acceptable to have 30 minutes lag. Other clients, however, may report their high watermarks differently.

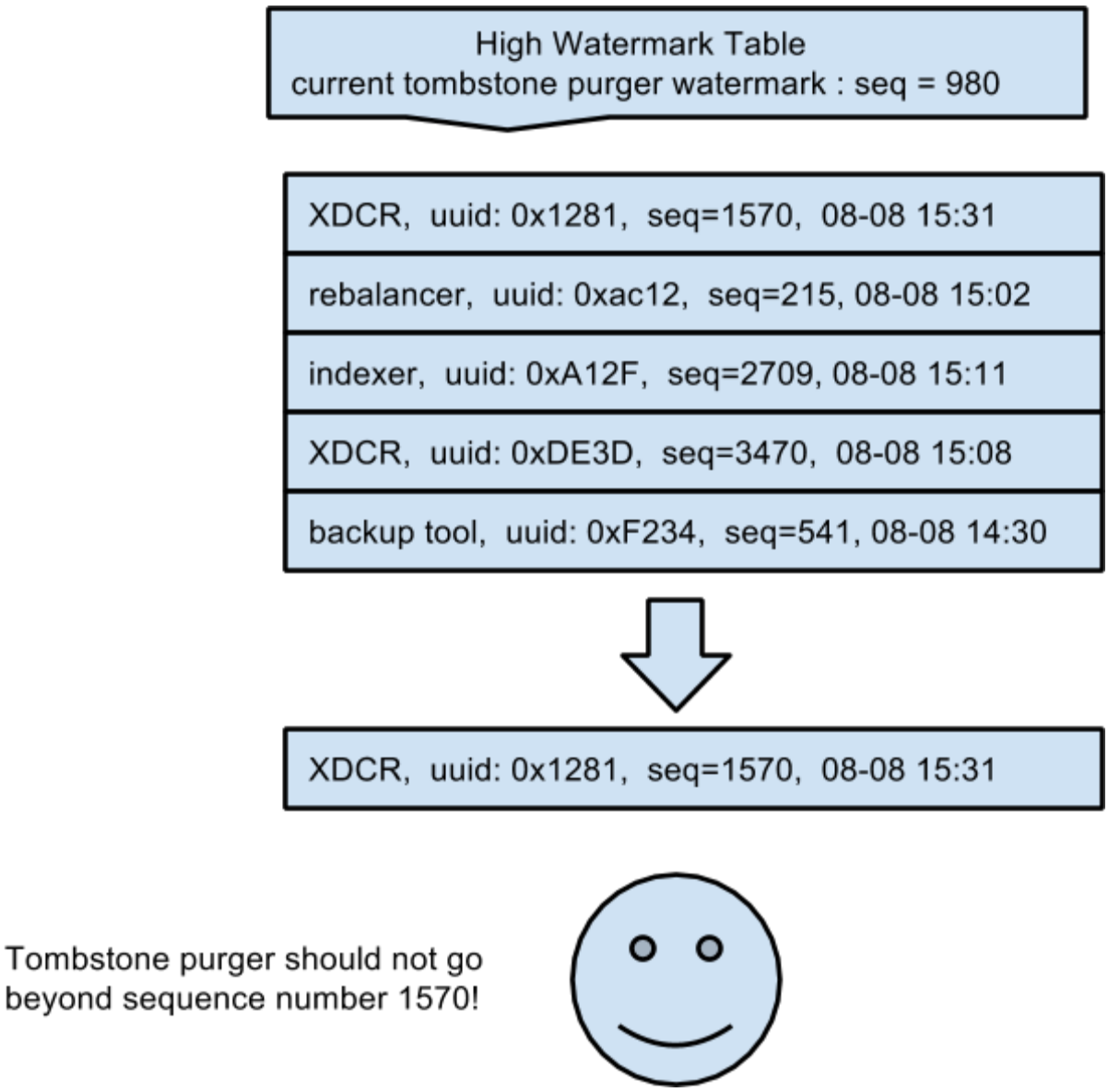
Workflow

When a UPR client starts consume mutations from scratch, it registers its water mark to HWT with a UUID. New watermark will override old watermark in HWT.



UPR client registers its high watermark to HWT when consuming mutations

If there are multiple UPR clients, over time there will be multiple entries in HWT. When compactor kicks in and tombstone purger need to determine the sequence when it can advance, it would [redacted] pick up a entry with smallest sequence number, which represents the slowest UPR client ahead of tombstone purger. The workflow can be shown in the diagram below.



Expiration of High Watermark

UPR clients may not be reliable. XDCR replicator may crash and restart due to different reasons e.g., loss of connection, topology change, etc. To avoid blocking tombstone purger, each entry of HWT is associated with an expiration time. Depending on clients, HWT will issue expiration time for each reported high watermark and expire it if the client does not update its high watermark before expiration. For example, if HWT expects XDCR to update its new high watermark in 1 hour, HWT can specify that each time XDCR reports its high watermark. Expired entry will be deleted from table when tomb purger computes next watermark to advance.

UPR client	Instance ID (UUID)	Reported High Watermark	Report Timestamp	Meta Logs	Expiration time in seconds
XDCR	5c576f8aa18	1008	2013-08-08T 9:37:19	XDCR checkpoint log	3600

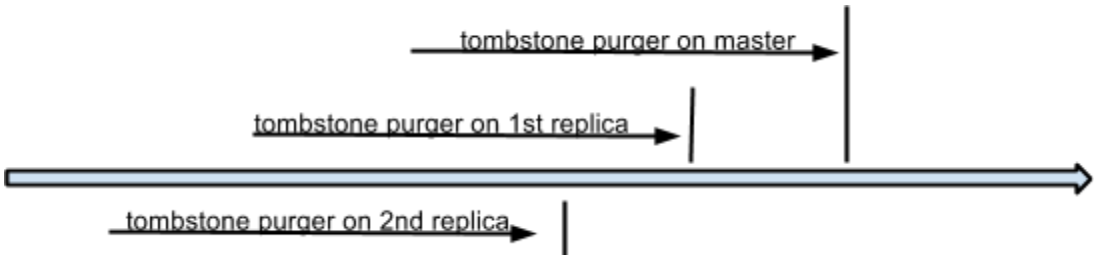
When a client comes back after its entry in HWT expired, the client need to check its own watermark and tombstone purger's current watermark. Since its entry has be removed, it is possible that tombstone purger has outpaced the clients, in this case, client has to start from the beginning at sequence number 0 to avoid missing deletions.

Intra-cluster replication of HWT

In order to ensure correctness when master server failover to a local replica, it is necessary to synchronize tomb purger on master and local replicas. Therefore HWT need to be replicated to local replicas from master from time to time. It is unclear what is the best update interval as the time of writing. At high level, frequent updating HWT from master to local replicas will make tombstone purger on local replicas "close to" that on the master, but this comes with higher overhead. Longer updating interval may result tombstone purger lag behind that on the master and thus bigger storage footprint at replica nodes. The correctness is ensured as long as 1) tomb purger on master is synchronous with that on local replicas; 2) replicating HWT is synchronous with local replication.

Synchronized tombstone purgers

To avoid the case that tombstone purger on replica moves faster that running on master, we need to synchronize tombstone purger on master and replicas. The purger on replica should be never beyond that running on master. For chain replication, the tombstone purger on each hop should be never beyond that running on the previous hop. For example, assume we have a master replicating to two local replica using chain replication, e.g, replicating from master to first replica, and from first replica to second replica.



Synchronized HWT replication

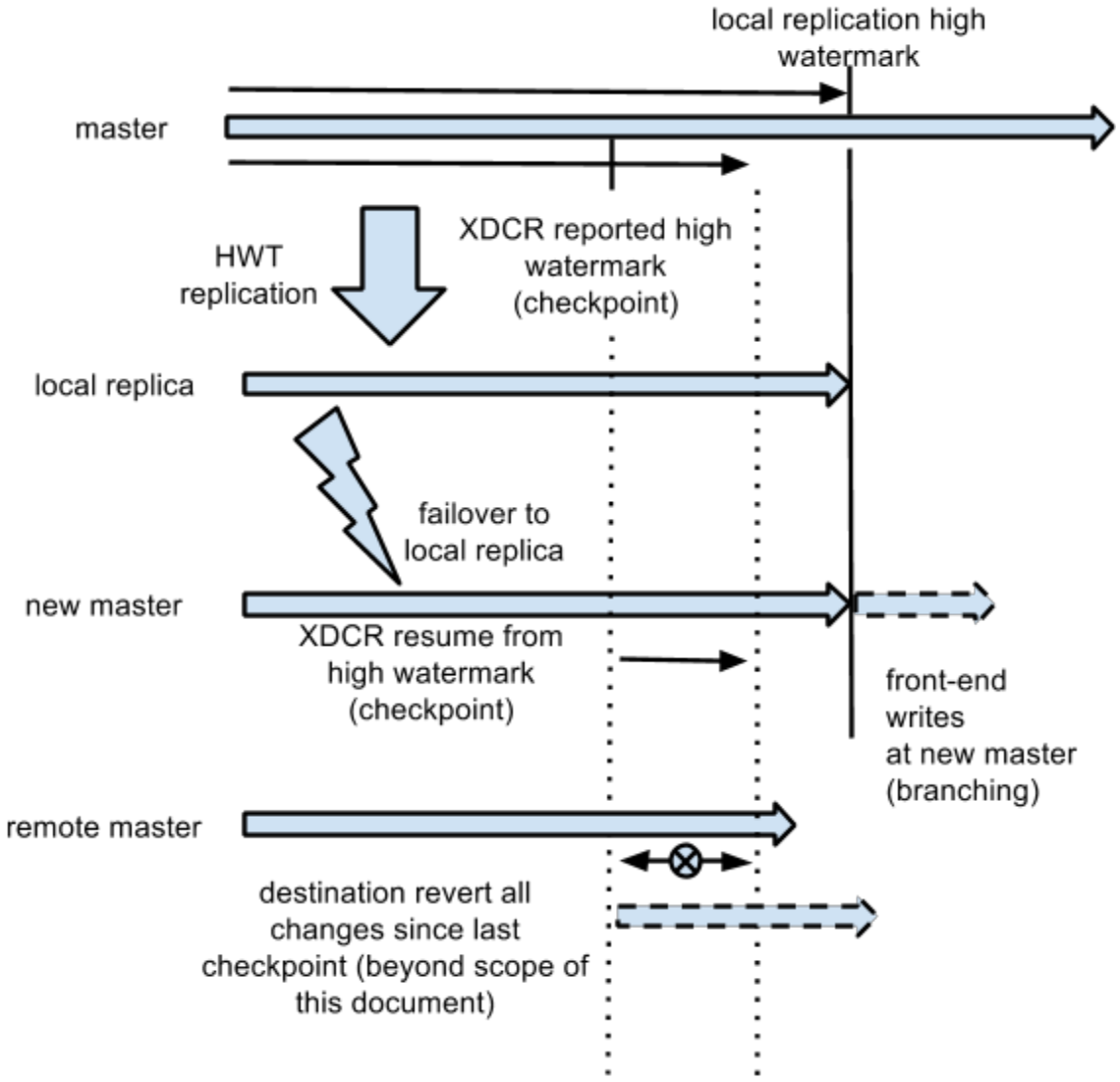
We should never replicate a HWT to local replica if there is a reported high watermark entry in HWT that is ahead of local replication high watermark. Asynchronous replication of HWT and local replication will result in data "gap" or phantom "future checkpoint". For example, if at a time the local

replication high watermark is 1024 while the XDCR has moved to beyond 2048, and record seq 2048 as the high watermark of
XDCR. Replicating such HWT is unsafe, because when failover takes place to the local replica, rejoining XDCR replicator will try to restart from seq 2048, which does not exist at new master (local replica) at all and XDCR misses all mutations between seq 1024 and seq 2048. For a chain local replication, if each hop enforces the same policy when replicating HWT to the next one, then we can fail over to any replica without missing any mutations when XDCR rejoins to the new master. If the local replication is using star topology, then each satellite replica need to be synchronized with master.

It is unclear whether other UPR clients such as indexer are subject to the same problem but at least remote replica (XDCR) need to be synchronized with local replication, and HWT naturally provides a vehicle to synchronize local replication and XDCR.

Example of Synchronized HWT replication

Synchronized HWT replication guarantees that when failover to a local master and XDCR can always resume from last replicated high watermark in HWT.

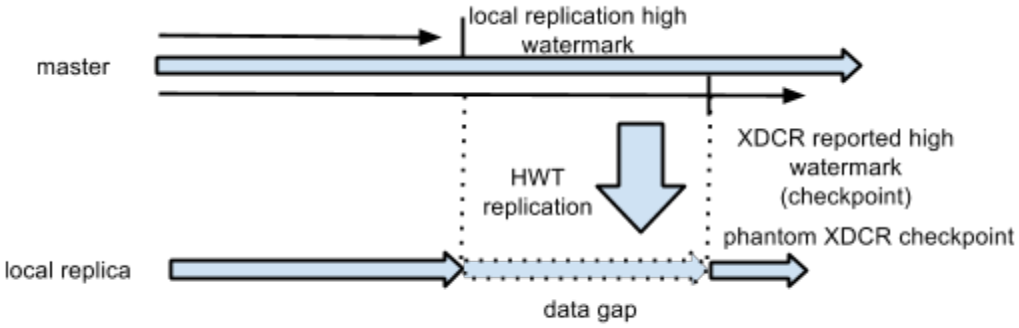


Synchronized HWT replication

Example of Asynchronized HWT replication

Asynchronized HWT replication may result in “data gap” when failover to a local replica and XDCR rejoin to

the new master after failover. In this case to ensure correctness, at destination side, XDCR need to discard all mutations which has been replicated, and re-start from scratch (seq = 0). The approach works only for master-slave XDCR where there is no front-end writes. Unfortunately it will not work in today's Couchbase architecture for master-master XDCR where there are front-end writes at destination. This is because we are unable to trace the source of mutations in Couchbase Server and thus unable revert all changes from XDCR while still keeping those from local front-end writes.



Data gap due to asynchronized HWT replication

Inter-cluster replication of HWT

Because today we never failover to remote replicas, HWT will not be replicated to remote clusters. In future we enable failover to remote clusters other than loca replica, we may need to enforce HWT inter-cluster replication to ensure correctness. This is beyond scope of this document.

References

1. Tombstone purger
2. UPR design spec