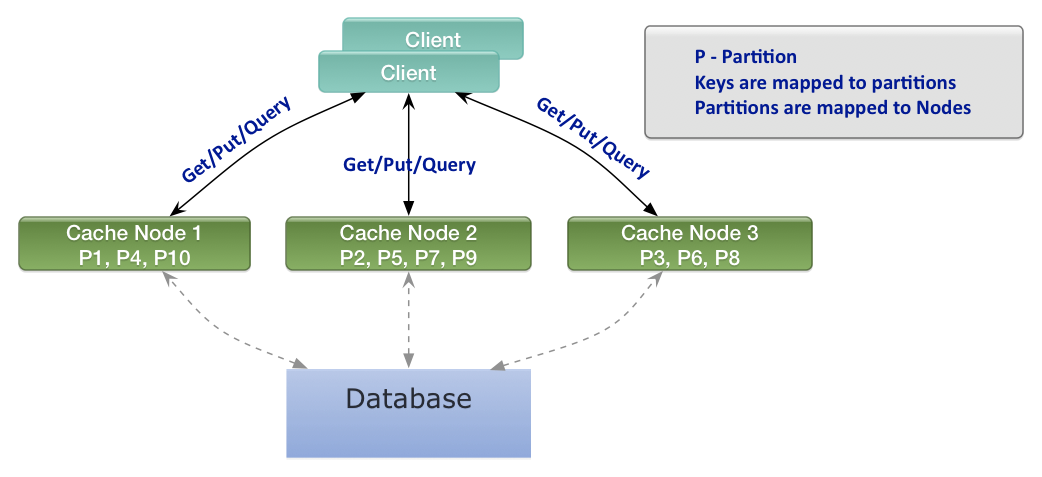
**[One-Phase-Commit - Fast Transactions For In-Memory Caches](http://gridgain.blogspot.com/2014/09/one-phase-commit-fast-transactions-for.html)**

In my [previous blogs](http://gridgain.blogspot.com/2014/09/two-phase-commit-for-in-memory-caches.html) I have talked at length about *2-Phase-Commit* transaction protocol for in memory caches, and how in-memory caches can handle failures a lot more efficiently than disk-based databases. In this blog I want to cover yet another very important optimization that can be utilized for in-memory caches, specifically for cases where data is partitioned across the network.  
  
  
In-memory caches, and specifically in-memory data grids, such as [GridGain](http://www.gridgain.org/) or [Oracle Coherence](http://www.oracle.com/technetwork/middleware/coherence/overview/index.html), often employ a technique called data partitioning, where every key in the key set is assigned to a partition, and every partition is assigned to a specific cluster member. Assigning keys to a partition is usually easy and is done similarly to how hash maps work:

***key.hashCode() % N***, where N is a total number of partitions.

Assigning a partition to a cluster node is a little trickier, as in case of failures or cluster topology changes, the amount of repartitioning has to be minimal. There are various algorithms that can be employed here, such as [Rendezvous Hashing](http://en.wikipedia.org/wiki/Rendezvous_hashing), or [Consistent Hashing](http://en.wikipedia.org/wiki/Consistent_hashing), which we will not be discussing here. Let's assume that after applying some of the partition strategy algorithms, your in-memory cache evenly distributed data among cluster nodes:

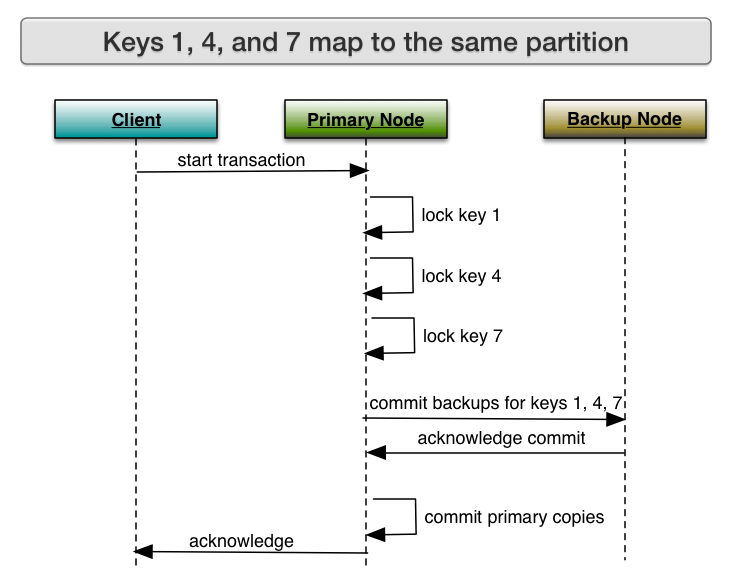
[](http://3.bp.blogspot.com/-sWj7nVBW9yY/VBfQlxorPcI/AAAAAAAACXQ/qJc3tbH-OD8/s1600/data-paritioning.png)

***Figure 1: Cache Partition Distribution***

Usually to achieve best performance we need to minimize the number of cluster nodes participating in a transaction. This can be done by ensuring that all the entries in the transaction belong to the same partition, which consecutively ensures that they all belong to the same node. It will also ensure that the backup copies for these entries will be grouped together on some other node as well (and secondary backups will be grouped together as well, and so on). Such custom key mapping is called ***custom affinity mapping***.  
  
For example, if we have *Employee* objects and *Company* objects, then we can ensure that all employees working for the same company will be mapped to the same partition by providing a custom *affinity-key* for *Employees*, which in this case will be the *"companyId"*.

***Custom affinity mapping helps us ensure that all objects within a single transaction are mapped to the same cluster node, thus collocating the whole transaction logic on that node and minimizing the number of nodes involved in a transaction.***

The *1-Phase-Commit* optimization is possible only when we have 1 primary and 1 backup copy. Such deployments are most common for distributed caches. If we add 2 backup copies, then we have to resort to *2-Phase-Commit*, however, adding a 2nd backup copy is generally considered wasteful from memory standpoint and is rarely done. Diagram below illustrates the *1-Phase* optimization:

[](http://1.bp.blogspot.com/-wsdtHOnteCc/VBjH3XI_4HI/AAAAAAAACYY/buQz8h09Cwc/s1600/1-phase-commit.png)

***Figure 2: 1-Phase-Commit for Collocated Partition Transaction***

The first deviation from standard transactions is that now the client node sends the whole transaction logic to the primary node. This is possible because we ensure in advance that all the keys we are transacting on are mapped to the same partition on that node. Once the primary node receives the transaction logic, it will acquire all the locks locally, and will send only one *commit* message (without the *prepare* message) to the backup node.  
  
Now let's analyze failure scenarios. In this case, failures of client nodes or backup nodes are not very interesting, as they do not affect the primary copies. Failures of primary nodes are a bit trickier, however they are still safe. If the primary node crashes before it sends the commit message, then the backup transaction never starts, so there are no side effects. If the primary node fails after or during the*commit acknowledgement* is received from the backup node, then the backup transaction is committed, and data consistency is again not violated.  
  
The hardest part is that even though the data remains consistent in case of any cluster failures, how does the client node know whether the transaction was committed or not if it failed to get the final *acknowledgement*, i.e. if the primary node failed before it was able to send the acknowledgement to the client? In this case the *"recovery protocol"* is initiated, which was described in detail in my [previous blog](http://gridgain.blogspot.com/2014/09/two-phase-commit-for-distributed-in.html). Essentially, a message is sent to the backup node asking whether the transaction was committed or not. Since the backup node keeps a backlog of completed transaction IDs in memory, it can always check the backlog. If the backlog does not have the given transaction ID, the backup node will add it in the *"rolled back"* state and will reply to the client with *rollback* acknowledgement. If afterwards the backup node actually does receive the commit request for the same transaction ID from the now failed primary node, it can verify in the backlog that it was rolled back and safely ignore it.

### Conclusion

By ensuring that all objects participating in a transaction are mapped to the same logical partition, we can remove the whole *"prepare"* phase from the distributed commit protocol, thus converting the standard *2-Phase-Commit* into very light weight *1-Phase-Commit*transactions

http://gridgain.blogspot.in/2014/09/one-phase-commit-fast-transactions-for.html