Implementation of MVCC Transactions for Key-Value Stores

ACID transactions are one of the most widely used software engineering techniques, a cornerstone of the relational databases, and an integral part of the enterprise middleware where transactions are often offered as the black-box primitives. Notwithstanding all these and many other cases, the old-fashion approach to transactions cannot be maintained in a variety of modern large scale systems and NoSQL storages because of high requirements on performance, data volumes, and availability. In such cases, traditional transactions are not rarely replaced by a customized model that assumes implementation of transactional or semi-transactional operations on the top of units that are not transactional by themselves.

In this post we consider implementation of lock-free transactional operations on the top of Key-Value storages, although these techniques are generic and can be used in any database-like system. In GridDynamics, we recently used some of these techniques to implement a lightweight nonstandard transactions on the top of Oracle Coherence. In the first section we take a look at two simple approaches that suitable for some important use cases, in the second section we study more generic approach that resembles PostgreSQL's MVCC implementation.

Atomic Cache Switching, Read Committed Isolation

Let's start with simple and easy-to-implement techniques that are intended for relatively infrequent updates in read-mostly systems, for instance, daily data reload in eCommerce systems, administrative operations like repair of invalid items, or cache refreshes.

The most trivial case is reload of all data in the cache (or key space). We wrap cache interface by a proxy that intercepts all cache operation like *get()* or *put()*. This proxy is backed by two caches, namely, A and B, and works in accordance with the following simple logic (Fig.1):

- At any moment of time, only one cache is active and proxy routes all user request to it (Fig.1.1)
- Refresh process load new data into inactive cache (Fig.1.2)

- Refresh process switches a global flag that shared by all proxies that participate in refresh and this flag defines which cache is active (Fig.1.3). Proxy starts to dispatch all new read transactions to the new active cache.
- Transactions that are in progress at the moment of cache switching can be handled differently depending on required level of consistency and isolation. If non-repeatable reads are acceptable (some transaction can read data partially from the old state and partially from the new one) then switch is straightforward and old data can be cleaned up immediately. Otherwise, the proxy should maintain a list of active transactions and route each one to the cache it was initially assigned. Old data can be purged only when all attached transaction were committed or aborted.

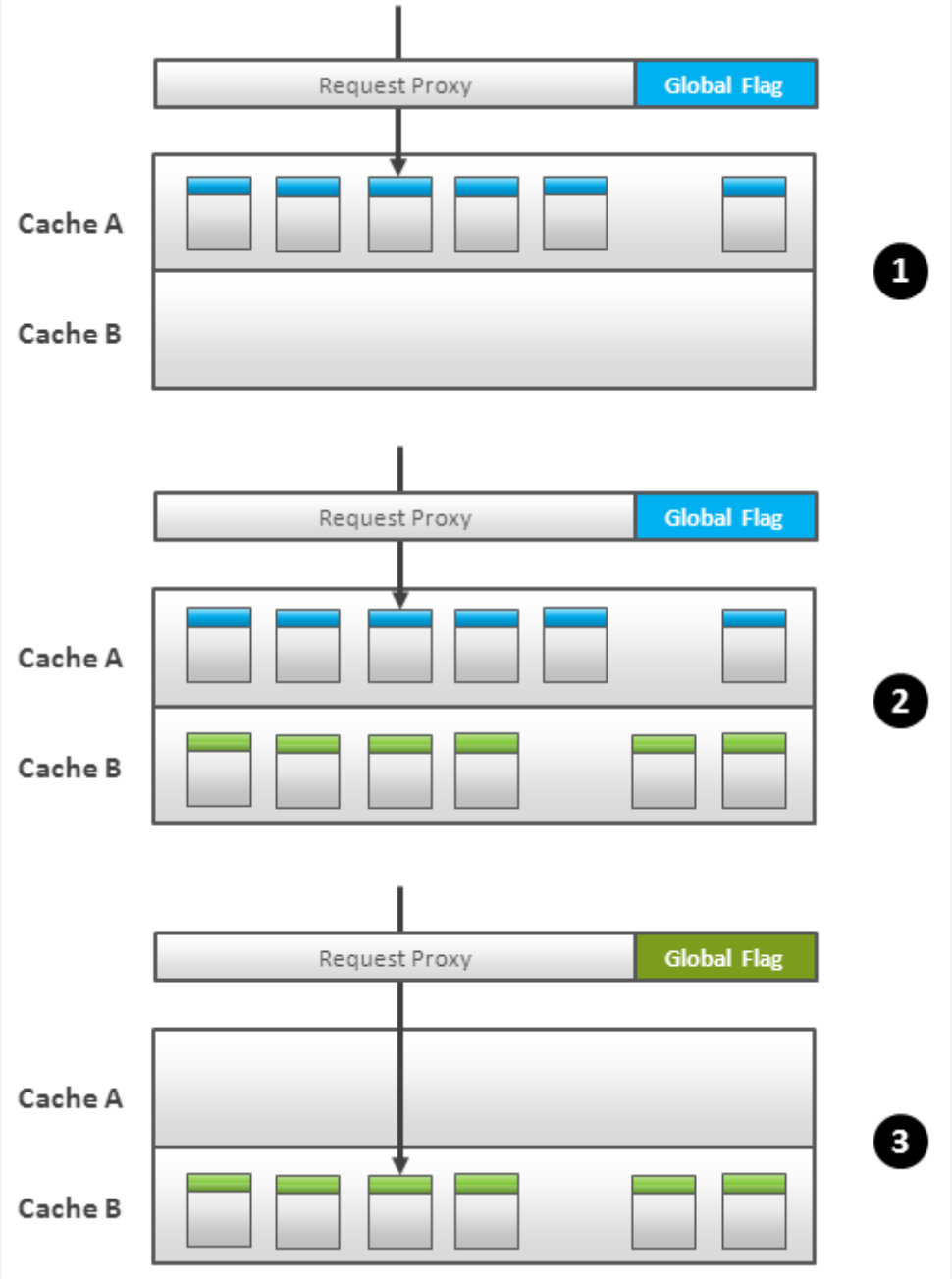

Fig.1 Cache Switch

The similar technique can be used for partial updates. It can be implemented differently depends on the underlying storage, we consider one simple strategy with three caches. The framework is similar to the previous one, but proxy acts in the following way (Fig.2):

- User requests are routed to the PRIMARY cache (Fig.2.1)
- New and updated items are loaded into the NEW cache, keys of deleted items are stored to DELETED cache (Fig.2.2)

- Commit process begins with switching of the global flag. This flag instructs the proxies to look up requested keys in NEW and DELETED caches first and, if not found, look up the same key in the PRIMARY cache (Fig.2.3). In other words, all user request are switched to the new data at this step.
- Commit process starts to propagate changes from NEW and DELETED caches to the PRIMARY cache, i.e. replace/add/remove items in the PRIMARY cache one by one in non-atomic way (Fig.2.4).
- Finally, the commit process switches the global flag back and requests are routed to the PRIMARY cache (Fig.2.5).
- Old data can be copied to another cache during step 4 in order to provide rollback possibility. In-progress transactions can be handled as for full refreshes.

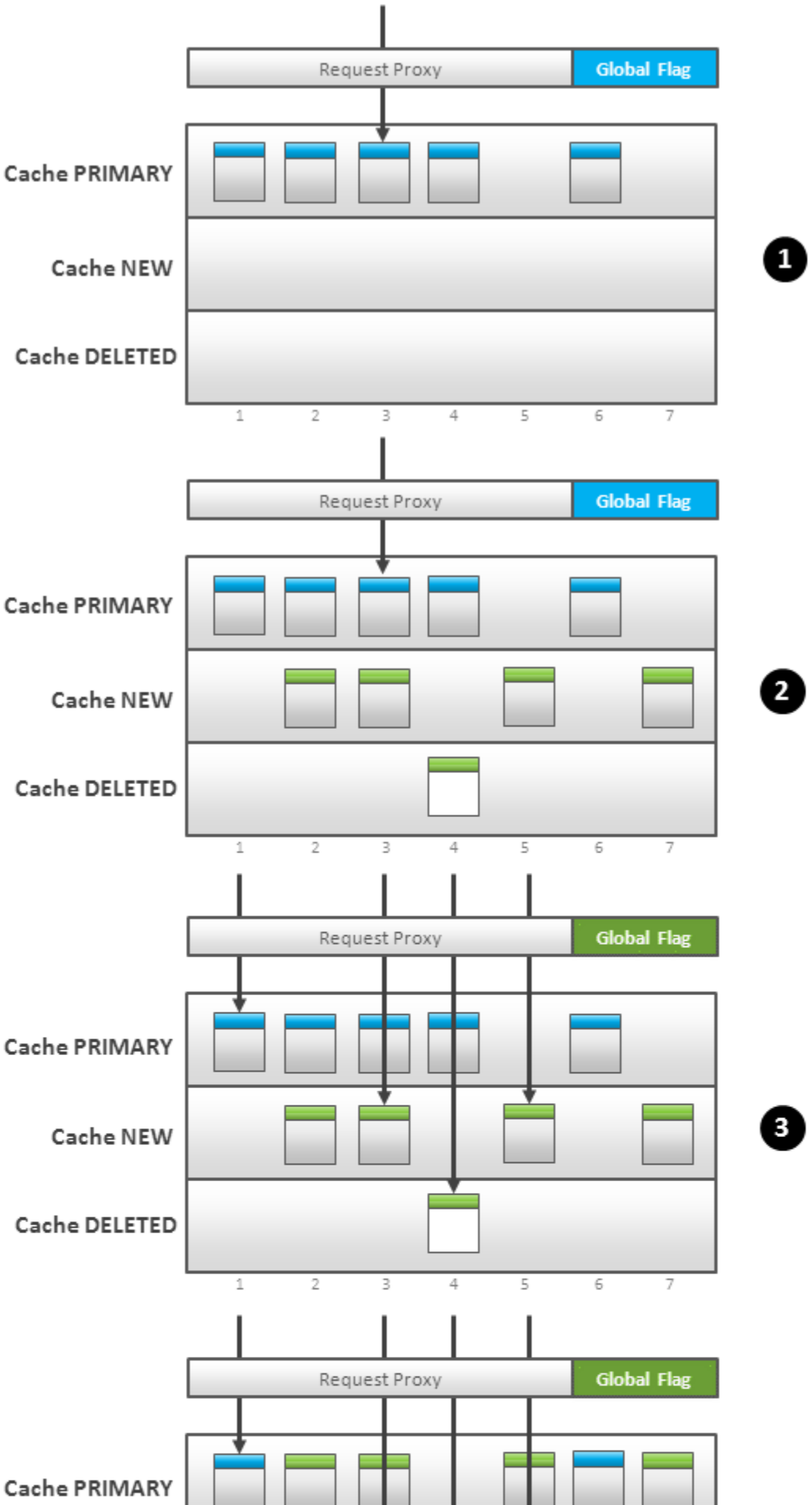
Request Proxy
Global Flag
Cache PRIMARY
Cache NEW
Cache DELETED
1
1 2 3 4 5 6 7
Request Proxy
Global Flag
Cache PRIMARY
Cache NEW
Cache DELETED
2
1 2 3 4 5 6 7
Request Proxy
Global Flag
Cache PRIMARY
Cache NEW
Cache DELETED
3
1 2 3 4 5 6 7
Request Proxy
Global Flag
Cache PRIMARY

Fig.2 Partial Cache Switch

Thus, from the examples above, we can conclude that attachment of read transactions to the snapshot of data and avoiding of interference from the commitment of the update transactions is one of the main sources of complexity. This is obviously a case for write-intensive environments. In the next section we consider very powerful technique that helps to solve gracefully this problem.

MVCC Transactions, Repeatable Reads Isolation

Isolation between transactions can be achieved using versioning of separate items in the Key-Value space. There are different ways to implement this technique, here we discuss an approach that is very similar to how PostgreSQL handles transactions.

As it was said in the previous section, each transaction should be attached to a particular data snapshot which is a set of items in the cache. At the same time, each item has its own life span – from the moment it was added to the cache till the moment it was removed or updated, i.e. replace by a new version. So, isolation can be achieved via marking each item two time stamps, each transaction by its start time, and checking that transaction sees only items that were alive at the moment the transaction began. In practice of course global monotonically increasing counters are usually used instead of time stamps. More formally:

- When a new transaction is started, it is associated with:
  - Its Transaction ID or XID which is unique for each transaction and grows monotonically.
  - A list of XIDs of all transactions that are currently in-progress.
- Each item in the cache is marked with two values, *xmin* and *xmax.* Values are assigned as follows:
  - When item is created by some transaction, *xmin* is set to XID of this transaction, *xmax* is empty.
  - When item is removed by some transaction, *xmin* is not changed, *xmax* is set to XID. The item is not actually removed from the cache, it is merely marked as deleted.
  - When item is updated by some transaction, old version is preserved in the cache, its *xmax* is set to XID; new version is inserted with *xmin*=XID and empty*xmax*. In other words this is equivalent to remove + insert.
- Item is visible for transaction with XID = *txid* if the following two statements are true:
  - *xmin* is a XID of the committed transaction and *xmin* is less or equal than *txid*
  - *xmax* is blank, or XID of the non-committed (aborted or in-progress) transaction, or greater than *txid*

- Each *xmin* and *xmax* can store two bit flags that indicate wherever transaction aborted or committed in order to perform checks described in the previous point.

  This logic is illustrated in the following graphic:

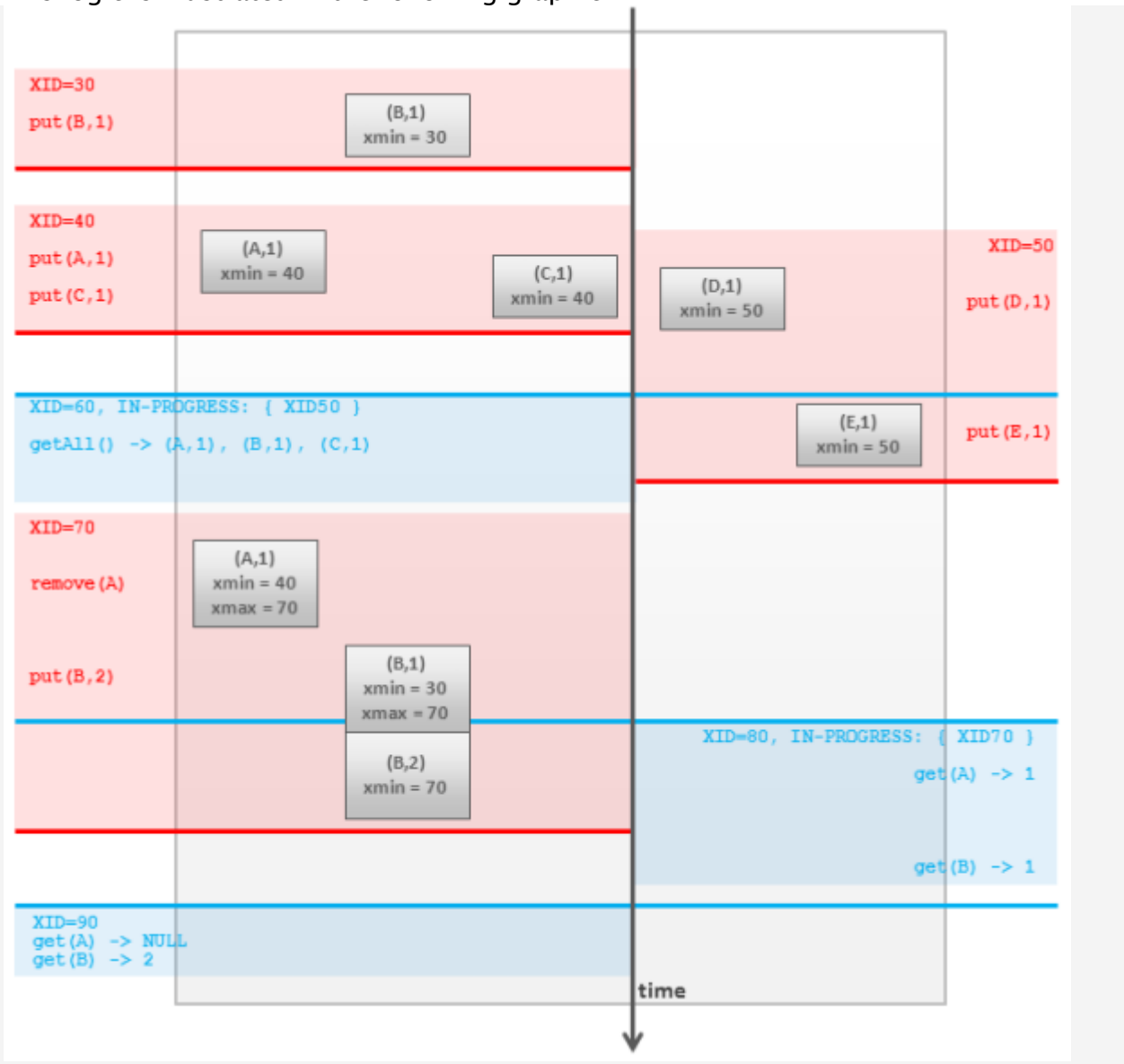

Fig.3 PostgeSQL-like MVCC

The disadvantage of this approach is a quite complex procedure of the obsolete versions removal. Because different transactions will have visibility to a different set items and versions, it is not straightforward to determine a moment when particular version becomes invisible and may be eliminated. There at least two different techniques to do this, the first one is used in PostgreSQL, the second one in the Oracle Database:

- All versions are stored in the same key-value space and there is no fixed limit on how many versions may be maintained. Old versions are collected by a background process that is executed continuously, by schedule, or triggered by reads or writes.
- Primary key-value space stores only the last versions, the previous versions are stored in another fixed size storage. The last versions have references to the previous versions and particular version can be traced back by transactions that require them. Because size of the storage is limited, oldest versions may be eliminated to free space for the "new old" items. If some transaction is not able to find a required version it fails.

Source: http://highlyscalable.wordpress.com/2012/01/07/mvcc-transactions-key-value/