Space Management

There is a “tree nodes” space of fixed size units, each the size of a tree node with a bit map to keep track of the free units.

There is an AVL tree to keep track of the free space available needed to store the Values of every KV.

* We need trees in RAM also. How are they to be handled? Do we keep space for “tree nodes” in RAM too? Or do we use malloc and mfree?
  + We need to keep track of free space in a RAM tree for efficiency (F\_AVL).
  + We need to keep a search tree in RAM (K\_AVL).
* In NVM, we are not doing any searching for items using the NVM K\_AVL. So, do we need it at all?
  + If we remove it, the meta-data will be in the WAL. But now the WAL will need to be periodically “cleaned” to remove deletes and updates, as otherwise it will keep on growing. So, at boot time, the NVM WAL will be used to create the RAM K\_AVL.
  + Alternatively, we could consider using a hash table (K\_hash) as an index in NVM. So, when an insert in K\_AVL, and after this is done, the new key is inserted in K\_hash). Same for a delete. The WAL remains as before.
* F\_AVL – Can we consider other data structures for keeping track of free space? Since Values are of variable length, we cannot use fixed size units and bitmaps. We can use a buddy system starting at 8 bytes.

Index Management

* Can we replace K\_AVL with K\_hash in RAM too? Search will be faster. But, range queries will not be possible.
* Suppose we target DBMS tables as our storage items. Each row of a table will be a KV, with the K being the primary key of the table.
  + Now, we break up each key into two parts: tableid || rowid (is a fixed size key a good idea now? Primary key of table may have different lengths. We can use an internal “row number” as a key. But then we will be losing the “free” index on the primary key. How do we handle this?)
  + We use a hash table on the tableid portion of a key to locate the root AVL tree of the particular table’s rows. We then search the identified tree for the key. Range queries across tables are not likely, so this scheme will handle range queries.
* DBMS tables need indexes on non-key fields too (so called seconday indexes). We can implement such indexes easily using hashing. We can also create secondary index trees for particular trees. But all these indexes have to be supported at the meta-data level directly, and not left to the user program to implement on top of our KV store.
  + This may lead to a new design which is not merely a KV store!
* We need to take into consideration the cost of doing joins of tables on our design.

Some slide presentations are included for you to get some more ideas.