Storage queue in FoundationDB is an in-memory, queryable structure that keeps 5s worth of versioned data on a storage server. It's commonly referred to as PTree in the FDB code base.

Data structure

Storage queue has been implemented using the 'Treap' data structure. It's technically a balanced binary search tree, but O(Log n) height is not guaranteed. It uses randomization and binary max heap property to maintain balance with high probability. Expected time complexity of search, insert and delete in a Treap is O(Log n).

Every node of Treap maintains two values: * Key: Follows standard BST ordering (left is smaller and right is greater) * Priority: Randomly assigned value that follows Max-Heap property.

Partial Persistence

Partial persistence in a data structure is the ability to read at any given version, but the ability to modify only the latest version. Storage queue in FDB offers partial persistence for 5s worth of versioned data. There are three common techniques to achieve partial persistence in a data structure: * Copy-on-write * Fat Nodes * Path copy * https://en.wikipedia.org/wiki/Persistent_data_structure

PTree has implemented partial persistence using a combination of Fat node and path copy technique. In this hybrid technique, one additional updated pointer is stored in the PTree nodes. And an update at a new version uses the updated pointer, if available. Else, it creates a new node and follows the path copy technique.

Node structure

PTree nodes are fast allocated (using the FDB fast allocator) and reference counted. Each node is separately fast allocated. i.e. there is no guarantee what arenas they will end using. Each node is fixed in size at 96 bytes. Nodes contains references to keys and values in the mutation log and an integer priority (and hence fixed size). Each node contains 3 pointers: left child, right child, updated child if any.

Whenever the 3rd pointer (i.e. updated child) in the node is occupied, that means the node contains an update at a newer version. It's important to note that the update actually happened to one of the child nodes (left or right) and not actually to the node. And the node will have the following special fields set: * Updated flag: Indicates that the node was updated (imp to look at this in addition to the updated child pointer. Because the child pointer might be present because of an uncleared node as well) * replacedPointer: Indicates which child (left or right) was actually updated * lastUpdateVersion: The version at which the update happened

Versioned Map

Versioned Map is the structure that's exposed to the storage servers. It's a std::deque of pairs of versions and corresponding PTree root nodes. Since the versions are monotonically increasing, the data is sorted intrinsically. So we don't need to store this information in an in-order data structure (such as std::map which is a red-black tree underneath).

Versioned map provides a logical abstraction of a separate PTree at any given version. Physically it's a single PTree which contains data across the 5s worth of versions.

It helps the storage server process find the root of the tree at a given version during various read operations. Whenever a new version is created, root node corresponding to the latest version is copied into the new entry and may get modified as operations are performed on the PTree.

Storage Queue Mutations and Operations

Storage queue provides many read-write operations to it's clients, such as the storage servers. Such as inserting new <key, value> pairs, clearing certain key-ranges and range reads.

Clear Range Mutation

This operation erases a [startKey, endKey) range at a given version. Note that the endKey of range is not-inclusive. Clear range is a two step process: * Step 1: Erase the <key, value> pairs in the given range from the PTree as of the given version * Step 2: Insert a special marker <startKey, endKey> (aka tombstone) indicating the presence of clearRange Tombstone insertion is necessary to have for readRange at the storage server level to work correctly. This is because, at the storage server level, the result of the readRange operation is the merge of read results from the storage queue and the storage engine. The presence of the tombstone ensures that we skip over that range while reading from the storage engine as well.

Set Mutation

This operation inserts a <key, value> pair at the latest version. The steps involved in inserting a <key, value> pair in the storage queue are as follows: * Locate the root of the PTree at the latest version * Traverse the PTree to find the correct position - typical BST way * Create the node with given <key, value> and attempt to insert at the desired position * Generate a random priority value and check if the max heap property for priority has been violated * Causes a rotation if violated

Presence of clearRange mutations needs special handling during Set mutation When inserting a key that happens to fall in the middle of a clearRange, we need to split that clearRange. e.g. Let's assume that there already exists a clearRange [A, E) in the PTree, now inserting C would cause the clearRange to split into: [A, C) and [D, E).

Read Range Operation

As mentioned above, read range operation has been implemented at the storage server level. As the name suggests, it returns values with-in the specified key-range. At storage server level, it's the merge of read range results from in-memory PTree and the storage engine.

Recall that PTree is a binary search tree under the covers and hence there is no super-efficient way to do a range scan (like a B+Tree where the leaf nodes are connected can do a blazing fast range scan).

Also note that the FDB PTree implementation doesn't store parent pointers, which might help going back to the ancestor. With these limitations, and with the goal of achieving efficient range scans, this operation uses an iterator with a finger vector: like a stack of nodes visited. This helps remember the ancestor in the absence of a parent pointer. This works well while moving forward and saves a bunch of work. But note that there is more wasted work while moving backwards.

Range read finds the low bound based on key range beginning and then can move forward or backward (depending on the direction of range read) using the finger vector of the iterator.

Interesting properties

Presence of clearRange mutations leads to some complexity in the readRange path. readRange gets split into multiple readRange requests if there are clearRange mutations present in the desired key-range. End bound of current readRange iteration is at the start of next clearRange. Then data is read from the storage engine for that (sub)range and the results are merged with the PTree read results.

PTree is traversed once to establish these end bounds and then again while reading the data between start bound and end bound in the merge step. This double traversal is important for efficiency and correctness in the presence of clearRange mutations. Because we don't want to read the key-range of a clearRange from storage engine. But it can be wasteful in the absence of a clearRange. We can potentially cache the results of PTree traversal while establishing the end bound and use that during the merge process.

Potential known issues with Storage Queue

Storage queue growth can be a concern in certain scenarios. If the storage queue gets backed up, it can have a negative impact on the system's overall performance.

Scenarios that can lead to storage queue backing up: * When the underlying disks are not able to keep up, and the data versions start to accumulate in the storage queue. * When there are relatively few data mutations, but large number of versions. This makes the storage queue large * In the current implementation, the storage queue keeps everything since last persisted version to very latest. Even if that corresponds to more than 5s worth of data versions * For a write heavy workload (any workload can have periods of high write rate, e.g. during initial or incremental loads), if the number of storage server processes is not configured properly (if it's less than the optimal value per disk), that can also lead to the storage queue being backed up. Because there won't be enough disk write requests and they won't reach their peak IOPS.

optimal value
of SS per disk?