華中科技大學

课程报告

课程名称：区块链技术与应用

专业班级：计算机硕 2001 班

学　　号：M202073236

姓　　名：甘泉

指导教师：肖江

报告日期：20201221

计算机科学与技术学院

目 录

1 Merkle tree 快速过滤（布隆过滤器）

1.1 Merkle tree 的介绍

Merkle tree（默克尔树）以 Ralph Merkle 命名，Ralph 在 1979 年对该数据结构申请了专利（于 2002 年过期）。在密码学和计算机科学当中，哈希树或 Merkle tree 是一种特殊的树结构，其每个非叶节点通过其子节点的标记或者值（子节点为叶节点）的哈希值来进行标注。哈希树为大型的数据结构提供了高效安全的验证手段（wiki）。在比特币网络中，Merkle 树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹，且提供了一种校验区块是否存在某交易的高效途径。生成一棵完整的 Merkle 树需要递归地对哈希节点对进行哈希，并将新生成的哈希节点插入到 Merkle 树中，直到只剩一个哈希节点，该节点就是 Merkle 树的根，如图 1-1 所示。

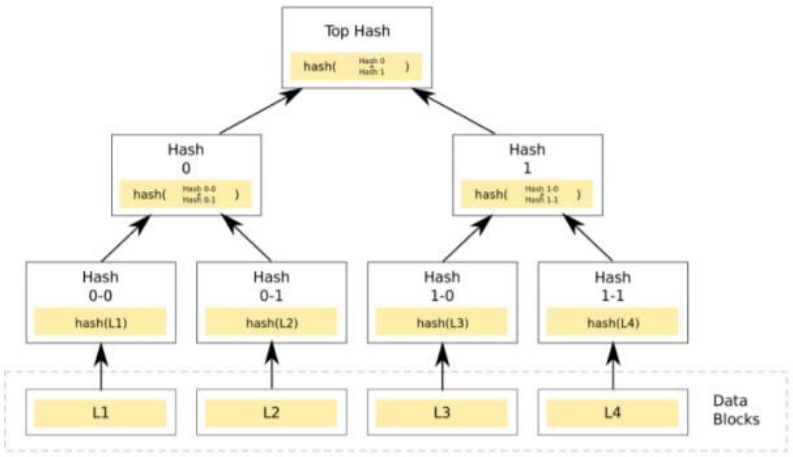

图 1-1 Merkle tree

如图 1-1 所示，Merkle tree 中的节点可分为两类：

① 叶子节点：在二叉树中，没有子节点的节点称为叶节点，这是初始节点，对于一个区块而言，每一笔交易数据，进行哈希运算后，得到的哈希值就是叶节点。

② 中间节点：子节点两两匹配，子节点哈希值合并成新的字符串，对合并结果再次进行哈希运算，得到的哈希值，就是对应的中间节点。

1.2 Merkle tree 在区块链中的作用

大部分普通人在电脑上安装的比特币钱包都是轻量级（非全节点）的（SPV，simple payment verification），只存储区块链头部。SPV 能够通过区块链头部中有限的信息证明某一笔交易是否存在于区块链中。

如图 1-2 所示，假设我们要判断 Tx3 交易的存在，只需要生成一个仅有 2 个哈希值长度的 Merkle 路径，该路径上有两个哈希值，Hash01 和 Hash2。由这两个哈希值产生的认证路径，再通过计算另外 2 个哈希值 Hash23 和 Merkle Root，任何节点都能证明 Hash3 包含在 Merkel 根中，即证明了 Tx3 交易的存在。

验证过程如下：

Step1：计算交易 Tx3 的哈希值，得到 Hash3。

Step2：通过 Hash2 和 Hash3 的哈希值，得到父节点的哈希值 Hash23。

Step3：同上，通过计算 Hash23 和 Hash01 哈希值，得到根节点的哈希值。

Step4：将上一步得到的根哈希值对比区块头中 Merkle Tree 的根哈希值，如果相同，则证明该区块中存在交易 Tx3，否则说明不存在。

使用 Merkle 树可以大大降低 SPV 节点的存储和计算负担。

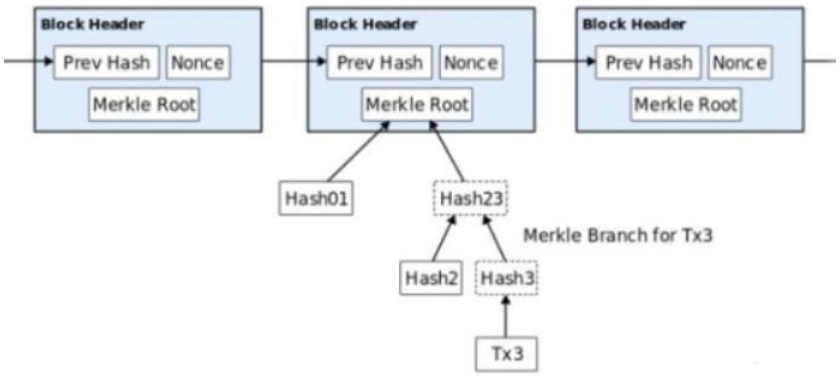

图 1-2 区块中的 Merkle tree

1.3 优化 Merkle tree 的出发点

由 1.2 中介绍的验证过程，在每次验证之前都需要对叶子节点进行遍历，然后在按照 1.2 节介绍的验证步骤，自下而上的沿着验证路径一次次的计算哈希值，直到计算出根哈希值，并最终与 Merkle Tree 记录的根哈希值进行比较，如果相同，则说明查找的数据存在。

在计算和比较哈希的过程中，任意一次的比较结果不相同即停止验证。

在验证 Merkle tree 是否包含某个项时，最耗时的操作是其中对叶子节点的遍历操作。尤其是当叶子节点数大时，对于叶子节点的遍历操作将成为主要开销。如果能够利用布隆过滤器，自上而下的对 Merkel tree 进行查找，避免遍历操作，在牺牲一定空间的基础上，便能够较快地对 Merkle tree 进行过滤。

1.4 优化方案

利用布隆过滤器快速检索一个元素是否存在于一个集合中的特性，可以实现 Merkle 的快速过滤。

在自下而上的创建 Merkle tree 时，为每一层的节点添加一个布隆过滤器，对于最底层的叶子节点而言，其布隆过滤器中就只包含有该项对应的 hash 值；对于树中的每一个内部节点而言，为了能够自上而下的进行过滤，其布隆过滤器中应该包含以该节点为 root 节点的树中所有节点的 hash 值。为了实现这一点，只需要计算其左右孩子节点的布隆过滤器“或”值即可。即假设某个内部节点中的布隆过滤器为 C 的左右孩子节点的布隆过滤器为 A 和 B，则 C=A|B。在自下而上的构造 Merkle tree 的过程中，为每一层的节点添加并计算对应的布隆过滤器，如图 1-3 所示。

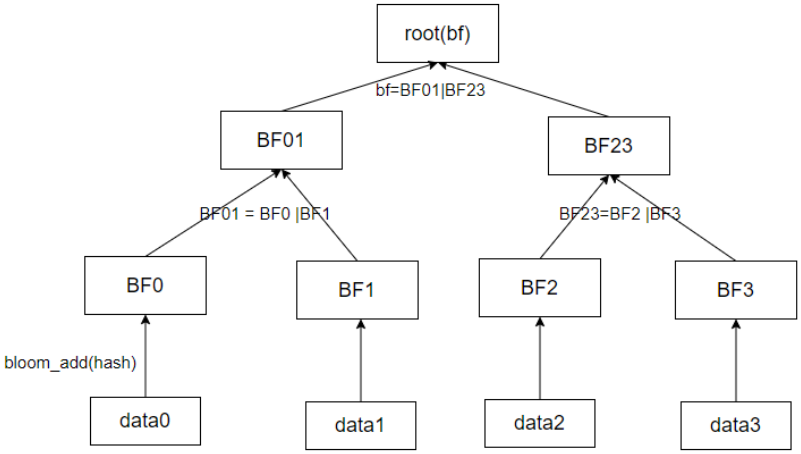

图 1-3 BF 的构造

在为每一个节点添加了布隆过滤器之后，对于某个项的验证过程如下，以 data0（hash0）为例。

Step1：计算数据项 data0 的哈希值 hash0

Step2：利用根节点 root 的布隆过滤器进行过滤，判断是否包含该 hash，如果包含，执行 Step3，否则，返回上一步骤，并向下执行。

Step3：如果该节点不存在孩子节点，则说明当前节点为叶子节点，验证成功。否则，执行 Step4。

Step4：在左孩子节点为当前节点，执行 Step2。

Step5：在右孩子节点为当前节点，执行 Step2。

Step6：执行到此步骤，说明验证不存在该数据项。

简单来说，验证过程即简化为了对一个有序的完全二叉树（可能存在复制节点）进行深度遍历，直到探测到某个符合条件的叶子节点，时间复杂度为 $O(\log_2 n)$。而普通的 Merkle tree 验证的时间复杂度接近于 O(n/2)。其中 n 代表树中所有节点数。

注意事项：由于叶子节点数并不一定是 2 的指数倍，所以，为了构造出一个完全二叉树，不可避免地需要对节点（包括中间节点和叶子节点）进行复制。当某层的节点数为奇数时，需要复制其中某个节点，用于构造上一层的节点。需要注意的时，由于在进行验证时，采取的是深度遍历，所以为了减少不必要的开销，需要对复制而来的节点进行标记，即使用这类节点的布隆过滤器进行过滤，因为总是会使用到被复制节点的布隆过滤器。

具体实现见代码及代码注释。

1.5 优化效果

测试环境如表 1-1。

表 1-1 测试环境

硬件配置	
CPU	Intel® Core™ i5-10210U CPU
内存	16GB
SSD	512GB
软件配置	
操作系统	WSL(ubuntu20.04)
IDE	VSCode

测试参数配置即对应的测试数据如表 1-2，其中 bloomfilter 的 entry 设置为 10000，特殊情况下跟随 ITEM_NUM，误差率设置为 0.01。

表 1-2 测试数据

ITEM_NUM	TEST_WITH_BLOOMFILTER/ms	TEST_WITHOUT_BLOOMFILTER/ms
512	0.621	1.304
1K	2.302	4.208
2K	4.625	15.098
4K	15.363	54.480
8K	22.782	314.702
16K	59.040	624.745
64K	406.880	6657.092

根据测试数据绘图如图 1-4 和图 1-5。

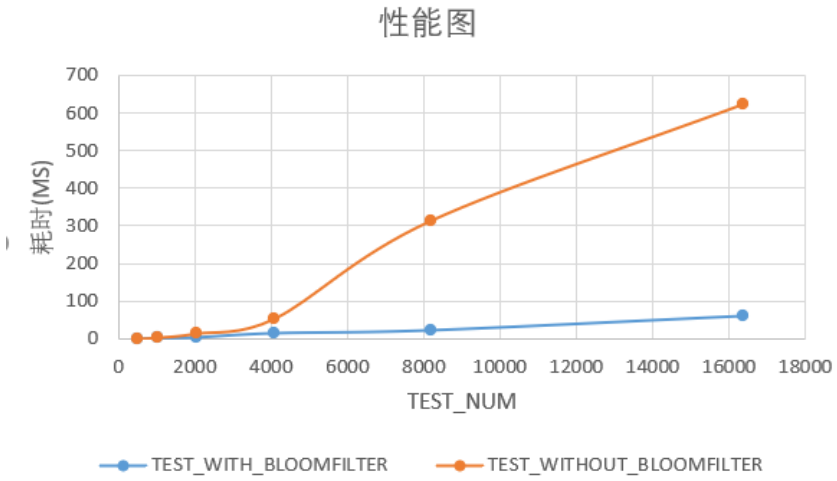

图 1-4 性能测试图

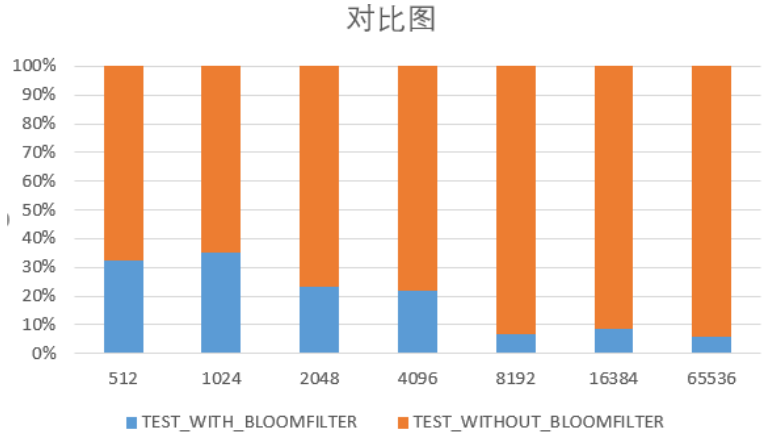

图 1-5 时间对比图

由图 1-4 可知，在所有测试情况下，使用布隆过滤器进行优化的 Merkle tree 的表现要明显好于不带布隆过滤器的。并且从图 1-5 可知，当测试基数越大时，带有布隆过滤器的优势要更加明显。

正确性验证见测试代码。

1.6 总结

代码仓库：https://github.com/Junkrat-boommm/MerkelTree-With-BloomFilter

利用布隆过滤器快速过滤的特性，在 Merkle tree 添加布隆过滤器，通过深度遍历的方式对树中的布隆过滤器进行验证，避免对叶子节点进行遍历，能够使得 Merkle tree 的过滤速度极大增大，大大降低其空间复杂度，但与之而来的是更长的树的构建时间以及更大的空间复杂度。