A Distributed Fault-tolerant Secondary Index on PBFT-based Block Chain

Yanchao Zhu[1], Zhao Zhang *[1,2], Peng Cai[1], Weining Qian[1], and Aoying Zhou[1]

[1] School of Data Science and Engineering,
[2] School of Computer Science and Software Engineering,
East China Normal University
{yczhu}@stu.ecnu.edu.cn,
{zhzhang,pcai,wnqian,ayzhou}@sei.ecnu.edu.cn

Abstract. Keywords:

1 Introduction

Recently, block chain is becoming more and more popular for de-centered application, such as Bitcoin, Ethereum and Hyperledger. These systems tolerate the Byzantine General Problem, which means they can guarantee the correctness of system even when these are several bad nodes in the system. In practice, these systems are indeed required for applications such as decentralized currencies, tracking use of charitable funds and Intelligent contract. In these systems, data is stored in database. These systems use consensus algorithm such as PBFT, PoW and PoS for consensus before querying or modifying the state of the system. Applications interact with the system using intelligent contract which is stored on each node(store procedure). The system must scan the whole data set distributed in order to answer queries on non-primary key attributes. With the increment of size of data, query performance will become worser. In practice, building a secondary index is a popular solution to decrease response time of the queries on non-primary key attributes. In block chain systems, all data are stored on each node, building indexes on a single node will occupy lots of storage space. Whats more, single node is not able to provide service for concurrent business query. To the best of our knowledge, there are no approach for distributed secondary index on block chain. The major difficulties include (1) (2)

We have made the following contributions in this paper:

- sssssssssss
- sssssssssssssssssss
- sssssssssssssssssss

The rest of paper is organized as follows: Section 2 introduces background knowledge related to our work. Section 3 gives an overview of index construction. Discussion of details of bulk loading is in Section 4. We then provide an evaluation of the approach in Section 5. Finally, we describe related work in Section 6, and conclude our paper in Section 7.

* Corresponding author.

2 Background

3 Index Structure

A common structure of an index is an index table, which is partitioned and stored on a cluster of nodes as an ordinary one. A record in the index table is combination of an index column *(searchkey)* and a primary key column of the data table, like as *(searchkey,primarykey)*. Figure 2 shows an example of the index. The primary key of the item table is column *ItemId*. If we create an index on the column *Sale*, the schema of index table is shown in Figure 2. A query is executed by accessing the index table to get the primary key of the data table, and then getting results from the data table according to the aforementioned primary key. Certainly, we also permit users to build the covering index, which is an index that contains all, and possibly more, the columns that you need for your query.We use a common partition strategy of hash for index partition.

4 Index Operation

4.1 Index Insertion

For index insertion with index value V, a node need to judge if it need to modify index table. If the node store the index of value V, it will update the incremental cryptography of index value V for query validation. Insert operation consists of steps shown in Algorithm 1: Line 6-judge if the node needs to store the index value. Line 7-add index entry into index table. Line 8 to line 9 -update the incremental cryptography of index value V. Since each node is store all data of the system. Modify of index will not cause distributed transactions which is important to a distributed system.

Algorithm 1: Index Insertion

```
1  Let H denote a hash map of incremental crypto of index value;
2  Let S denote value of key V in hash map;
3  Let N denote the number of index partition;
4  Let ID denote the number of node;
5  Function IndexInsertion(index_value,rowkey)
       /* local index construction                                    */
6      if ID == hash_value(value) then
7          add_index(value, rowkey);
8          S = H.get(value);
9          S.add_incremental(rowkey);
10     end
11 end
```

4.2 Index Deletion & Update

For index deletion with index value V, a node need to judge if it need to modify the index table. If the node store the index of value V, it will update the incremental cryptography of index value V for query validation. Insert operation consists of steps shown in Algorithm 1: Line 2-judge if the node needs to store the index value. Line 3 to Line 4-delete index entry from index table. Since each node is store all data of the system. Modify of index will not cause distributed transactions which is important to a distributed system.

Algorithm 2: Index Deletion

```
1 Function IndexDeletion(index_value,rowkey)
    /* delete index                                          */
2   if ID == hash_value(value) then
3     delete_index(value, rowkey);
4     S = H.get(value);
5     delete_incremental(S, rowkey);
6   end
7 end
```

5 Query

In order to get correct data, the client need to send a query request to the block chain, which means a query request need N 2 times of network communication. By adding the trusty node, a query just need 2 times of network communication. The query operation of the index consists of three part: query route, index query and query validation.

Algorithm 3: Index Update

```
1 Function IndexUpdate(old_value,new_value,rowkey)
    /* update index                                          */
2   if ID == hash_value(old_value) then
3     delete_index(old_value, rowkey);
4     S = H.get(old_index_value);
5     S.delete_incremental(old_index_value);
6   end
7   if ID == hash_value(new_index_value) then
8     add_index(new_value, rowkey);
9     S = H.get(new_index_value);
10    S.add_incremental(rowkey);
11  end
12 end
```

Algorithm 4 describe the process of query route, the trusty node will send the request to the node storing the index entry. The node storing the index

entry will search local index and return the result along with the incremental cryptography, the trusty node will check the cryptography for validation.

When a node receive a request for query of index, it will search local index for query. Since index is distributed on several node, each node maintain a partition of index, the query process is efficient. Line 2 to line 6 in Algorithm5 describe the process of query on a node.

Algorithm 4: Index Query

```
1  Function IndexQuery(value)
       /* query by index                                          */
2      if ID == hash_value(value) then
3          S = H.get(value);
4          V = get_rowkey(value);
5      end
6      send_result(value, V);
7  end
```

Since a node storing an index partition may be faulty, the trusty node will check the response from the node. If the cryptography of result is not the same as the result on the trusty node, the trusty node will search local index for response. The validation process handled in the following way: the trusty node will first calculate the cryptography of the return result of an index value, then it will compare the cryptography with the cryptography of its own for the value. If the two cryptographies are same, the result is correct, else, the result is wrong. Line 2 to line 3 of Algorithm6 describe the process of validation and line 7 to line 10 describe the process of searching local index of the trusty node when the return result is wrong.

As we can see, the trusty node just need to check the result and search local index when the result is wrong.

Algorithm 5: Query Validation

```
1  Function QueryValidation(value, V)
       /* validate result of query result                         */
2      S = H.get(value);
3      correct = validate(S, V);
4      if correct then
5          return_resultV;
6      end
7      else
8          V = search_local_index(value);
9          return_resultV;
10     end
11 end
```

6 Trusty Node

In a PBFT-based system, at most n/3 nodes are faulty. Since index is distributed on several nodes, nodes storing index partitions may be faulty, it may not return result, or return error message. In order to get correct result by index, a naive approach is first to construct index on each node, and then handle a query with the PBFT protocol. However the approach need to store index on all nodes and each query need n2 times of network communication. In our approach, we use a trusty node for data validation. For each query, the trusty node checks the result from index partition. If the result is correct after validating, the trusty node will return result,else it will search its own local index for response. By adding a trusty node, we avoid acquiring result by PBFT protocol which takes much network resource, which means our approach is more efficient.

Since the trusty node need to check the result returned by other nodes, it must be contain all index value. For each insertion of index, the PBFT cluster can apply change to the state machine if and on if the change has been made to the trusty node. The property of insertion guarantee the trusty node to contain all the changes to the state machine. In order check the result of query quickly, we use the technology of incremental cryptography. For each index entry, we maintain a cryptography for it, for modifications of index, we modify the cryptography for that index entry. For validating the result of query by index, the trusty just need to calculate the cryptography of returned result instead searching local index. If the cryptography of returned result is the same as the cryptography on trusty node, the result can be returned to client, else the trusty node need to search local index for response.

6.1 Combination of PBFT & RAFT

As is mentioned before, in order to check the returned result of query, the PBFT cluster can not apply changes to state machine before receiving acknowledgement from the trusty node. However, if the trusty node crashes, the system will block until recovery of the trusty node which is intolerant. Thus, the system need to provide service even when the trusty node crashes. Consider the situation in a blockchain, some nodes are high-level and some nodes are common, common nodes may send wrong messages for profit while high-level nodes are trusty and do not send wrong messages. Consider the property, we divide nodes in a blockchain into two partitions, on partition use the consensus protocol of PBFT and other nodes use the protocol of RAFT. The division of nodes in a blockchain on the one hand can improve the performance of blcokchain, on the other hand, since high-level nodes are trusty and can be organized as a RAFT cluster, it can play the role of trusty node in section 6.

In order to guarantee the consistency of the state machine on each node, we need to combine nodes using protocol of RAFT and nodes using protocol of PBFT. In the following of this section, we will show the protocol combine PBFT and RAFT.

6.1.1 The Algorithm The key point of combining PBFT with RAFT is to apply changes to PBFT and RAFT in an atomic way. In order to achieve the goal, we extend PBFT to commit if and only if RAFT has applied changes. We extend RAFT leader to send log according to the sequence number in PBFT, the leader og RAFT does not send out an log until it has received enough commit info from PBFT.

The algorithm works roughly as follows:

7 Experimental Evaluation

8 Related Work

9 Conclusion

References

1. Apache HBase website. `http://hbase.apache.org/`.
2. CDEAR website. `https://github.com/daseECNU/Cedar/`.
3. LevelDB website. `http://leveldb.org/`.
4. OceanBase website. `https://github.com/alibaba/oceanbase/`.
5. PHOENIX website. `http://phoenix.apache.org/`.
6. Secondary Index for HBase. `https://github.com/Huawei-Hadoop/hindex`.
7. SOLR website. `http://lucene.apache.org/solr/`.
8. Sysbench website. `http://dev.mysql.com/downloads/benchmarks.html`.
9. S. Alsubaiee and A. et al. Asterixdb: A scalable, open source bdms. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.
10. E. Brewer. Pushing the cap: Strategies for consistency and availability. *Computer*, 45(2):23–29, 2012.
11. F. Chang, J. Dean, and G. et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
12. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Conference on Symposium on Opearting Systems Design & Implementation*, pages 107–113, 2004.
13. P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
14. W. Tan, S. Tata, Y. Tang, and L. L. Fong. Diff-index: Differentiated index in distributed log-structured data stores. In *EDBT*, pages 700–711, 2014.
15. Y. Zou, J. Liu, S. Wang, L. Zha, and Z. Xu. Ccindex: A complemental clustering index on distributed ordered tables for multi-dimensional range queries. In *IFIP International Conference on Network and Parallel Computing*, pages 247–261. Springer, 2010.