Multi-tenant Main Memory Index Tree with Shared Structure

Lida Zou¹, Qingzhong Li², Lanju Kong³
¹,³School of Computer Science and Technology, Shandong University, Jinan, Shandong 250101, China
²Electronic Commerce Research Center of Shandong University, Jinan, Shandong 250101, China
*Corresponding author, e-mail: zouli@163.com, 1qz@sdu.edu.cn

Abstract
Multi-tenant main memory index is an important tool to accelerate data access to software as a service. Establishing main memory indexes for each tenant occupies lots of memory space and results in performance bottleneck. The data schemas and access patterns of different tenants are similar, which provides the conditions for tenants storing their index entries with shared structure in main memory. In the paper, the designed structure of main memory index puts the indexes of different tenants on one tree to achieve the effects of saving memory space. Meanwhile, tenant placement algorithm in the cloud is proposed. Its places the tenants with similar indexes on the same node to further optimize the main memory space.

Keywords: Multi-tenant database; Main memory index; Tenants placement; SaaS

Copyright © 2016 Universitas Ahmad Dahlan. All rights reserved.

1. Introduction
With the fast development of cloud computing, many Software as a Service (SaaS) [1, 2] providers deploy tenant applications in the cloud. The excellent multi-tenant database is the basis for multiple tenants to quickly access the data. With the falling cost of main memory, tenant data and indexes are gradually placed into main memory, which reduces the times of disk I/O and speeds up data processing.

The indexes in main memory are critical for quick and correct access to multi-tenant data. In multi-tenant database, many tenants share the processing resources. If indexes are separately built for each tenant in the scarce memory, access performance isolation is achieved but lots of memory space is occupied. It further leads to memory resource starvation. Ablut et al. [3] experimentally validate that the multi-tenant database with share schema has a high performance. The schema is also widely used in industry field [4]. Due to the similarity of tenant business, multi-tenant database with share schema usually build indexes for tenants on the same attributes of a data schema. This provides the conditions for the sharing of multi-tenant indexes in main memory.

There are several challenges in the sharing of multi-tenant memory indexes. First, it is difficult to differentiate the ownership of index entries since index entries of many tenants are put on the same index tree. Especially when a tenant does a range query, many index entries of other tenants are scanned and large amounts of useless data are processed. Second, when different combinations of many tenants share the main memory indexes, there are various space efficiencies. In the cloud, the node with share-nothing structure is used to process multi-tenant data. Thus choosing the tenants with the similar indexes for each node can improve space usage efficiency. However, it is a complicated and time-consuming task to make a choice among massive tenants.

To address the above issues, we propose a multi-tenant shared memory index tree (MSM). The tree is improved based on ART [5] and a multi-tenant placement algorithm in the cloud is given as well. Our main contributions are as follows.

The proposed MSM saves space and avoids the access to useless data. Experiments show that MSM saves space at least 30% than the benchmark algorithm when more than three tenants share the index tree and its data access performance reduces less than 15%.
The proposed tenant placement algorithm put the tenants with the similar indexes on the same node. Experiments demonstrate that it saves space 20% than the benchmark solution.

The rest of the paper is organized as follows. Section 2 introduces the related work. In Section 3 we give the structure of multi-tenant shared memory index and its related operations. Section 4 discusses a multi-tenant placement algorithm based on central tenants in the cloud. Experimental results are presented in Sections 5 and 6 concludes the paper.

2. Related Work

For the shared schema of multi-tenant data, index management is an important part of efficient querying. Weissman et al. use a series of Pivot Tables to store tenants’ indexes in the Force.com cloud [1], which achieves the quick access to tenant data. Aulbach et al. [6, 7] propose multiple shared schemas of multi-tenant data and design the methods of building indexes on Chunk Table. It divides tenant data into two kinds and respectively stores them in two tables depending on whether tenants have the demands of building indexes. The indexes are built on one table and the other table is not indexed, which guarantees isolation for tenant indexes in lower resource cost. The above studies give the solution to index isolation problem, and ignore the performance bottleneck caused by too many indexes stored in the main memory.

In the aspect of main memory index, many algorithms such as ART [5], Red-black trees [6, 7] and T-trees [7] are proposed and widely used, where ART performs better in both performance and space efficiency. ART is an adaptive radix tree for efficient indexing in main memory and each node in the tree dynamically adjusts its space size according to the number of keys. ART saves the space, but does not discuss the main memory index structure of multi-tenant data. In the paper we propose a main memory index tree shared by multiple tenants. The tree is based on ART, and saves the space through sharing an ART by tenants.

3. Multi-Tenant Shared Main Memory Index

For the shared schema of multi-tenant data, index management is an important part of efficient querying. Weissman et al. use a series of Pivot Tables to store tenants’ indexes in the Force.com cloud [1], which achieves the quick access to tenant data. Aulbach et al. [6, 7] propose multiple shared schemas of multi-tenant data and design the methods of building indexes on Chunk Table. It divides tenant data into two kinds and respectively stores them in two tables depending on whether tenants have the demands of building indexes. The indexes are built on one table and the other table is not indexed, which guarantees isolation for tenant indexes in lower resource cost. The above studies give the solution to index isolation problem, and ignore the performance bottleneck caused by too many indexes stored in the main memory.

In the section we first give the related definitions of multi-tenant main memory index and then show the structure and operations of the index tree.

Let \( t_n, k = 0, 1, 2, ... \) denote a tenant and \( I_i, i = 0, 1, 2, ... \), denote an index. Multi-tenant database builds several indexes for a tenant and the index set of a tenant is denoted as \( I_{m_{k}} = \{I_j | I_j = m_k \} \). If different tenants build the index on the same attribute of a schema, we call it index overlap. If \( I_j \cap I_{m_{k}} \neq \emptyset \) stands and there exists an overlap between index \( I_{m_{k}} \) and index \( I_{m_{k}} \), we get \( I_i = I_j \).
When several indexes overlap, they are put into a MSM. The structure of MSM is shown in Figure 1. Each node in the tree includes ART data and tenant privilege information. Assuming that \( o_j \) is a tree node in MSM, the ART data of \( o_j \) is generated referring to literature [5]. The privilege information of \( o_j \) is used to describe which tenants the subtree rooted by the tree node belongs to. The privilege information of \( o_j \) has \( n \) bits, where \( n \) is the number of tenants using the index. We use \( b_i \in \{0,1\}, i = [0, n - 1] \) to denote a bit of tenant privilege information. Mapping function \( F(tn_k) \) is created to get the location of privilege information for the tenant according to tenant ID. If \( F(tn_k) = b_i \) holds, \( b_i = 1 \) represents that there are index entries belonging to tenant \( tn_k \) on the subtree rooted by \( o_j \), while \( b_i = 0 \) represents that the index entries with \( o_j \) as an ancestor do not belong to tenant \( tn_k \). The correspondence between tenants of mapping function \( F(tn_k) \) and the expressed locations is stored in the main memory.

Next we give the operations of tenant \( tn_k \) on MSM and assume \( F(tn_k) = b_i \).

**Point query.** The goal node is searched from root node in MSM according to ART querying algorithm. If leaf node is reached and its \( b_i \) value is 1, the data storage address is returned. If the \( b_i \) values of the nodes on the search path are 0, the failure information of querying is returned. Point queries just need to determine the value of \( b_i \) when searching each node. Thus its efficiency is the same with ART.

**Range query.** Assume the query range is \((a, c)\). The main steps are as follows.
1. Point query is used to search the value \( a \). We set \( o_j \) as the leaf node when searching the value \( a \) or the tree node whose \( b_i \) is 0 when the search stops.
2. Find the parent node \( o_p \) of \( o_j \). If the \( b_i \) value of \( c \) is 0, go to step (8).
3. Put all the child nodes of \( o_p \) in reverse order into a stack.
4. Pop a tree node \( o_q \) from the stack. If \( v_{o_q} c \) holds, it ends, where \( v_{o_q} \) is the minimal key value of \( o_q \).
5. If \( o_q \) is leaf node and its \( b_i \) value is 1, the data storage address is returned.
6. If the \( b_i \) value of \( o_q \) is 0, the tree node is abandoned; else all the child nodes of \( o_q \) are put into a stack in reverse order.
7. If the stack is not null, go to step (4).
8. If \( o_p \) has parent nodes, do \( o_j = o_p \) and re-execute step (2).

Range query saves searching the tree nodes not belonging to \( tn_k \) through the 6th step and thus the query efficiency is improved. Taking a query of range \((A3, A5)\) in Figure 1 as an example, let the location expressed by tenant privilege information be \( b_i \), i.e., the location of the blue arrow in Figure 1. When traversing in the pre order, since \( b_i = 1 \) of node \( A0 \) stands, its child leaf nodes store the index entries of the tenant. Then continue to search \( A0 \). Because the
node $A0$ satisfies $b_1 = 0$, its child nodes are not searched any more. Similarly, the search reaches $A2, A3$.

**Insertion and Deletion.** When a tenant inserts or deletes a key, it refers to the operations of ART algorithm and updates the corresponding bits of tenant privilege information. If the splitting and merging of tree nodes are involved, the bits of tenant privilege information for the new node are the join operation on the bits of the child nodes.

**Bulk Load.** In the cloud, data engine usually migrates tenant data from one node to another node in consideration of loads and security. When tenant data are migrated, their main memory indexes are migrated as well. Assume that the goal computing node has an index $I_i$, and the data of the migrated tenants has an index $I_j$. If $I_i = I_j$ holds, all the index entries belonging to the migrated tenants in $I_j$ needs to be bulk loaded into $I_i$, which realizes the sharing of index tree for the goal computing node. The intuitive bulk-loading method is to first export all the index entries belonging to the migrated tenants in $I_j$, then transfer them to the goal computing node and last insert them into $I_j$. The method needs to first export and then import, whose time efficiency is low. In the paper we give a bulk load scheme based on privilege information to improve its loading efficiency. Let $M$ denote the set of the migrated tenants, i.e., the data and indexes of the tenants in $M$ need migrating. The steps of index $I_j$ bulk loaded into $I_i$ are as follows: (1) Transfer $I_j$ to the goal computing node. (2) Choose each leaf node in $I_j$. (3) Let $o_q$ denote one leaf node. Examine the privilege information in $o_q$, and determine whether the index entry in $o_q$ belong to a tenant in $M$. (4) If the index entry belongs to the tenant in $M$, insert the index entry in $o_q$ into $I_i$ and change the privilege information of $I_i$. In Section 5, experiments will validate its efficiency.

Multiple tenants share the main memory based on MSM, and operate the index entries separately without influencing the other tenants. Next section we extend MSM in the cloud according to the characteristics of tenant indexes.

4. Multi-Tenant Placement Algorithm Based on Central Tenants in the Cloud

Multi-tenant database generates a series of indexes according to the query demands of each tenant. If the tenants on the same computing node have a high overlap ratio in indexes, lots of cache spaces are saved. Overlap ratio is the number of the overlapped indexes among different tenants. Therefore, a placement algorithm of tenants in the cloud is desirable to let each computing node have a high overlap ratio, as shown in Figure 1.
The index set of tenant \( t_{n_k} \) is denoted as \( I_{t_{n_k}} = \{ I_i \mid t_{n_k} \subseteq I_i \} \) and the tenant set is denoted as \( T \). If the number of tenants served by each computing node is \( m \), the problem is to solve the subsets \( U_i \mid i \in [0, \frac{|T|}{m} \cap] \) of \( \frac{|T|}{m} \) tenants when the number of tenants in each subset is no greater than \( m \). These subsets make \( \left\lfloor \frac{|T|}{m} \right\rfloor \) be minimal, where

\[
U_i = \bigcup_{m_i \in V_i} I_{t_{n_k}}
\]

The intuitive way of getting the optimal solution is enumerating, but its time complexity is \( O(|T|^k) \). In the paper we propose a heuristic algorithm called central tenants grouping algorithm. Its main steps are as follows.

1. Choose the tenant with the most indexes, denoted as \( t_{n_k} \).
2. Put the index set \( I_{t_{n_k}} \) of the current tenant into the subset \( U_i \) and try to put the index sets of the other tenants in turn into \( U_i \) in order to maximize \( |U_i| \) and let the number of tenants in \( U_i \) be no greater than \( m \).
3. Choose the central tenant \( t_{n_h} \) in \( U_i \), i.e., the overlap ratio between the index set of \( t_{n_h} \) and that of \( U_i \mid t_{n_h} \subseteq U_i \) is the highest. It is formulated as

\[
t_{n_h} = \arg \max \left( |U_i - I_{t_{n_h}}| + |I_{t_{n_h}}| - |U_i| \mid t_{n_h} \subseteq U_i \right)
\]

4. Repeat step (1), and do \( E \) iterations to generate the subset \( U_i \).

```
Input: \( T, m, \{I_{t_{n_k}} \mid t_{n_k} \subseteq T\} \), the times of iterations \( E \)
Output: \( \{U_i\} \)
1. \( k = \frac{|T|}{m} \) if \( m \) is the number of subsets;
2. FOR \( i \) from 0 to \( k - 1 \) DO
3. \( t_{n_k} = \arg \max \left( |I_{t_{n_k}}| \mid t_{n_k} \subseteq T \right) \);
4. FOR \( e \) from 1 to \( E - 1 \) DO
5. \( U_i = I_{t_{n_k}} \cap T \cap - \{t_{n_h}\}, V_i = \{t_{n_h}\} \);
6. FOR \( j \) from 1 to \( |T| \) DO
7. \( U_i = U_i \cup I_{t_{n_k}} \cap V_i \cap - \{t_{n_j}\} \);
8. IF \( j > m \)
9. \( t_{n_l} = \arg \min \left( |U_i - I_{t_{n_l}}| + |I_{t_{n_l}}| - |U_i| \mid t_{n_l} \subseteq V_i \right) \);
10. \( U_i = U_i - I_{t_{n_l}} \); \( V_i = \{t_{n_l}\} \);
11. END IF
12. END FOR
13. \( t_{n_h} = \arg \max \left( |U_i - I_{t_{n_h}}| + |I_{t_{n_h}}| - |U_i| \mid t_{n_h} \subseteq V_i \right) \);
14. IF \( t_{n_h} = t_{n_k} \) THEN BREAK;
15. END FOR
16. END FOR
17. Return \( U_i \mid i \in [0,k-1] \);
```

Figure 1. Central tenants grouping algorithm

Multi-tenant Main Memory Index Tree with Shared Structure (Qingzhong Li)
(5) Repeat step (1) to generate the remaining subsets.

Algorithm 1 gives the concrete steps. Its time complexity is $O(|T|^2E)$ and has a higher execution efficiency. In section 5.2 Algorithm 1 will be experimentally proven to better save main memory space.

5. Results and Discussion

In this section, we first test the space occupation and access performance of MSM, and then validate the influence of central tenants grouping algorithm on main memory space in the cloud.

We use OpenStack to build our experimental environment. Each computing node has a 64-bit Ubuntu system, 4 CPUs, 8G memory, 200GB of storage space. The networks of the computer nodes are built by high-speed switches with a speed of 9120Mpps. The number of virtual nodes is from 2 to 16. The cache capacity of each node is set to 8G. We choose the mode structure of TPC-C [8] to generate tenant data. Multi-tenant data processing is online and transactional, such as the customer relation management product in Salesforce. TPC-C is an OLTP base a project simulating goods management environment of a wholesaler and can be deployed on a multi-tenant data platform. Each tenant has 8 indexes and $10^6$ tuples, and index entries are randomly generated for each index [9-15].

5.1. Space Occupation and Access Performance of MSM

Figure 2 gives the comparison on space occupation of MSM and NS. NS is the method of creating independent indexes for each tenant using ART. As seen in Figure 2, for 3 tenants MSM occupies 30% less space than NS. It is also observed that the occupied space of NS grows quickly with the increasing tenants. However, the occupied space of MSM ascends slowly, because the more the tenants are, the more the overlapped index entries are [16].

![Figure 2. The space occupation with the number of tenants](image)

![Figure 3. The access performance with the number of tenants](image)
Figure 3 shows the comparison on data access performance of MSM and NS. MSM-P and NS-P denote their point queries, respectively. The data reveal that the data access times of the two index structures are 2ms or so. MSM-P and NS-P have the similar access efficiency. MSM-R and NS-R represent the range queries of the two index structures. The figure shows that MSM-R has longer query time but not exceeding 15% of that of NS-R. When the number of tenants is small, the time of range query in MSM is similar to that in NS. When the number of tenants is large, the query time of MSM ascends, since MSM needs to check the privilege information of each node. Because the range query method of MSM could avoid traversing the leaf nodes not belonging to the tenant, its query time does not increase significantly with the expanding number of tenants.

Figure 4 gives the performance of bulk loading scheme based on privilege information. We compare the proposed bulk loading scheme based on privilege information (BLPI) with the intuitive loading method (ILM). ILM first exports the index entries belonging to the migrated tenants on the original node, then transfer them to the goal computing node and last insert them into the shared index tree in turn. The experiments demonstrate that BLPI has more advantages on the bulk loading time, especially when the number of tenants is large. This is because BLPI avoids inserting the index entries of all the tenants on the index tree, but does the insertion in the granularity of leaf node.

5.2. Space Occupation of Central Tenants Placement Algorithm

Figure 5 demonstrates the influence of Central Tenants Placement algorithm (CTP) on the occupied space. We compare CTP with Random Placement Algorithm (RDP). In cloud computing environment, the number of tenants each computing node holds is 8, and the number of computing nodes ranges from 2 to 16. It is seen that CTP performs well on the occupied space when the number of nodes is large.
6. Conclusions

In the paper we propose a multi-tenant main memory index tree with shared structure to address the issue of multi-tenant indexes occupying too much memory space. The tree stores the multiple indexes built on the same attribute on one index tree, which achieves the effect of space savings. The operations of the proposed tree are given, whose performance reduces no more than 15% than that of the isolated indexes. A multi-tenant placement algorithm in the cloud is presented as well. It puts the tenants with high index overlap ratios on a computing node to further reduce the occupation of main memory space.

Acknowledgements

This work is supported by National Natural Science Foundation of China under Grant No. 61572295, No. 61303085; Natural Science Foundation of Shandong Province under Grant No. ZR2013FQ014, No. ZR2014FM031; Science and Technology Development Plan Project of Shandong Province No. 2014GGX101047; Shandong Province Independent Innovation Major Special Project No. 2015ZDJQ01002, NO. 2015ZDXX0201B03.

References


