The Implementation of B+ Trees Within MongoDB

Team 2 – Changxun Li, Haoxiang Geng, Yueqin Li

April 25, 2024

1 Abstract

As the world of data develops, many different SQL and NoSQL databases have been around each with their own speciality and purpose. One of these databases is MongoDB, the most popular NoSQL database on the market. After diving into its implementation, we found that MongoDB is mainly used in OLTP transactions, and it currently only supports a B-tree index, making us wonder what would happen if a fully-fledged B+ Tree index model was used instead, and how the performance would be enhanced in OLAP queries using B+ Tree. Hence, we dived into the MongoDB code base and implemented the indexing mechanism to become a B+ Tree.

Using the TPC-H benchmark we found that our new novel B+ Tree implementation outperforms the original B-tree by 1% - 25% in our chosen TPC-H scenarios, making it superior for certain OLAP workloads. The result also effectively validates the assumption that by switching to a fully-fledged B+ Tree indexing structure, we can boost the performance of scan queries that emphasize efficient scanning of data within a specific range and guide the search to relevant leaf nodes quickly.

2 Introduction

In database management, indexing plays an important role in increasing query performance and data retrieval efficiency. Currently, MongoDB utilizes a B-tree indexing mechanism (though a semi-B+ tree in its essence. For convenience, we'll refer to this structure as B-Tree, following their practice) within its default storage engine, WiredTiger. In scenarios involving sequential scans, cursor movements to adjacent pages necessitate traversing upwards to the parent node to ascertain the next or previous page, based on the hierarchical information stored at the parent level. However, this upward traversing behavior may not be the most efficient way to search the target.

Drawing from database theory learned in class, compared to traditional B-Tree we know that the B+ tree architecture offers superior performance for sequential scans and range scans. This enhancement stems from the implementation of a doubly linked list connecting the leaf nodes, enabling direct traversal to succeeding or preceding pages via next or previous pointers at the leaf level. This method significantly streamlines sequential access compared to the conventional B-tree approach. For this reason, we've decided to fully implement a B+ tree with a doubly linked list connection between the leaf nodes in MongoDB.

In this project we were able to first dive into how MongoDB handles indexing with WiredTiger as well as why they decided to do this, and then modify the source code binary to add additional links on top of the original B-Tree to facilitate the building and completion of a B+ Tree. Then using our newly constructed indexing algorithm, we conducted testing using TPC-H datasets and selected queries to compare the before and after performance of the database. Finally, we gave reasons regarding why some things work, and give recommendations based on the database's needs.

3 Proposed Solution

As mentioned before, we've decided to implement a B+ Tree with a doubly linked list to replace the traversal mechanics in MongoDB's original design. In this section, we'll first explain the indexing

mechanism in MongoDB's default storage engine – WiredTiger – where the original design resides. After that, we will introduce our methodology by first going over the B+ Tree's features which serves as background information, then expounding on our implementation of the B+ Tree. Finally, we mentioned some of the challenges we met during the implementation.

3.1 WiredTiger Exploration

As MongoDB operates on a document-based model, distinct from traditional relational databases, we must understand how MongoDB manages and stores data. This exploration aims to link our current insights on storage models in relational databases and the mechanisms employed by NoSQL databases like MongoDB. Our focus will extend to analyzing MongoDB's existing B-tree indexing mechanism to comprehend its workflow in handling queries. This analysis will inform us how MongoDB leverages its b-tree index in query execution and also serve as a foundation for potentially modifying our B+ tree index, ensuring it aligns with the workflow of the current indexing system.

At the core, B-Trees are constructed from internal and leaf pages. Internal pages act as directories, holding keys and pointers to guide searches toward the correct sub-tree. Leaf pages, on the other hand, hold the key-value pairs(key refers to the keys for indexing, and value refers to the pointer to actual documents). Each internal page utilizes a page index, essentially an array of pointers to its child pages, enabling efficient navigation during search operations. The code delves into the intricacies of traversing and manipulating these indexes during critical operations like page splits and cursor movements, ensuring the B-Tree's structural integrity.

WiredTiger B-Trees employ a navigation system where parent nodes act as guides, directing searches towards the correct child nodes and ultimately leading to the desired data. Internal pages within a B-Tree serve as the navigational core, holding keys that delineate the boundaries of the data contained within their child pages. Imagine an internal page with the key "M" and two child pointers. The left child would house keys falling below "M" in the sorted order. In contrast, the right child would contain keys greater than or equal to "M". This strategic division of key space allows searches to quickly determine which path to follow by comparing the search key with the keys in the internal node.

3.2 B+ Tree Overview

The indexing structure that we rely on – B+ Tree is a balanced tree data structure optimized for systems that read and write large blocks of data, like databases and file systems. It's designed to minimize disk I/O operations, efficiently managing large volumes of data by keeping data sorted and enabling quick insertion, deletion, and search operations. Typical features of B+ Tree included:

- Balanced Tree: All leaf nodes are at the same level in a B+ tree. This balance ensures that the time complexity for search operations remains consistently logarithmic
- Node Structure: Each node in a B+ tree contains a certain number of keys. Internal nodes have keys and pointers to their children, while leaf nodes contain keys and pointers to actual records or data. In a B+ tree, data pointers are only present in the leaf nodes.
- High Fan-out: A B+ tree has a high branching factor (fan-out), which means that each internal node has a large number of children. This reduces the height of the tree and thus reduces the number of disk reads during operations.
- Order of Tree: A B+ tree of order n can have at most n-1 keys and n children for each internal node. The root node is an exception, which can have a minimum of 2 children.
- Leaf Nodes Linked: The leaf nodes of a B+ tree are linked to each other in a linked list manner. This allows for fast and efficient in-order traversal and range queries.

3.3 B+ Tree Implementation in MongoDB

3.3.1 Modification in B+ Tree page structure

In a traditional B+ Tree, leaf nodes will contain a next pointer and a prev pointer to point to the next leaf page and the prev leaf page. Here we simulate the traditional B+ Tree implementation and add it into our solution. We modified the struct defined inside the btmem.h, which contains the definition of struct __wt_ref, and this struct further includes __wt_page , which is the actual pointer to pages. This means, both pointers can be used to locate a specific page, however, the __wt_ref will provide more information than __wt_page. We also found that, inside the definition of __wt_page struct, it includes a parent pointer, which is __wt_ref type, pointing to its parent internal node. Therefore, we decided to add the next and prev pointers into the __wt_page struct and also defined them as __wt_ref type.

After modifying the structure, we need to initialize the pointer as the page is created. The bt_page.c is used for the memory allocation of pages, and specifically the __wt_page_alloc function is used for creating or reading a page into the cache, where the page will be further accessed by DBMS. Therefore, we initialize the next and prev pointers inside this function.

3.3.2 Implementation of linked list between leaf nodes

After defining and initializing the next/prev pointers inside the leaf nodes, we need to figure out how to connect them and build up the linked list between them. Here we decide to connect the pointer during the data insertion process. In this process, as the data is inserted, it periodically splits the existing leaf node into several nodes, since the number of entries of nodes in B+tree is strictly defined.

MongoDB inserts the data in two ways: single insertion and bulk-data insertion. Accordingly, inside the bt_split.c, there are two functions called __split_insert and __split_multi. The __split_insert function implements the process of single record insertion, and the __split_multi function corresponds to the bulk-data insertion process. The __split_insert function is defined as splitting a page's last insert list entries into a separate page. We simply connect the original page and the newly created separate page by using the next/prev pointers. The __split_multi function is defined as splitting a page into multiple pages, which means when a batch of data is inserted, the original page might need to split into multiple pages to store all the records inside them. The newly created pages are stored in an array called ref_new. Similar to the connection process inside __split_insert function, we used a for loop to iterate this process, and handle the leftmost and rightmost nodes as edge cases, as they are related to NULL pointers.

Note that the original implementation of WiredTiger B-Tree uses __wt_page type to indicate the newly created leaf nodes. However, inside the bt_walk.c, which implemented the traversal on the tree(we will talk about it later), the cursor on the tree is defined as __wt_ref type, which means that, if we want to move the cursor using our newly created next/prev pointer, which is also __wt_ref(this definition correspond to the cursor definition inside bt_walk.c), we need to re-write the original __split_insert and __split_multi function, modified the definition of the newly created pages inside them, to align with both our definition of pointers and the implementation of bt_walk.c.

3.3.3 Utilizing the linked list for traversal

As we mentioned before, the bt_walk.c file implements the traversal of B-Tree. The B-Tree is traversed by iteratively moving the cursor on the nodes. This is implemented by the core function __tree_walk_internal, which controls the movement of the cursor and jumps to the next/prev node. The traversal is done by iteratively calling this function. In the original implementation, the movement towards the next/prev node is implemented by: (1) recursively ascending to the parent node until the cursor does not reach the last slot of the parent node (2) entering the next entry inside the parent node (3)keep descending until we find the correct leaf node. To utilize our pointers, we skip this complicated process by directly moving the cursor based on the next/prev pointer of the current node. This modification, to our thoughts, would decrease the number of operations during the process of traversal, hence enhancing the efficiency.

3.3.4 Implementation Challenges

The implementation process is not as trivial as it may look, and we encountered many challenges:

- Hard to understand the code base and find out where to modify: WiredTiger's code base is huge. Rather than a simple B+ Tree implementation which only has hundreds of lines of code, WiredTiger's B-Tree implementation has more than 30 files, and each file has thousands of lines of code on average. Also, since it is written in C, many variables and structs are defined inside header files, which brings additional challenges in understanding the code base.
- Modifying existing implementation and utilizing our proposed solution: Since WiredTiger uses B-Tree as indexing, it does not consider adding the functionalities of B+ Tree. Thus, simply adding our modification to the code might not work due to the logical conflicts between each file(e.g. the bt_walk and bt_split conflict mentioned before). This conflict brings additional challenges to re-writing the existing code to accommodate our modification.
- Hard to debug: As we were implementing our proposal, we found out that it was hard to find out whether our implementation was on the right track since MongoDB's log doesn't help a lot. Therefore, we wrote a lot of additional debugging code to print the intermediate results, even the whole B-Tree structure, to make sure that we did not make mistakes. This process also brings additional challenges through the whole process.

4 Experiments

4.1 Setup

For our test machine, we are using a 2021 M1 ARM-based Macbook Pro running macOS 14.4.1 with 16 GB of memory. For this experiment, we are going to be using MongoDB v6.0 and WiredTiger v10.0.2 as both the base system and the systems that we are modifying. For the testing data, we are using data generated by the TPC-H benchmark which is a widely recognized industry standard benchmark designed to evaluate the ability of a system to handle analytical OLAP workloads. We also decided to use Q1, Q6, Q12, Q14 from the official TPC-H documentation as our testing queries, we first had to translate them from the original SQL format into MongoDB NoSQL then translate it again into Python to use with PyMongo as our connector to use MongoDB.

We choose Q1, Q6, Q12, Q14 because these queries showcase several key characteristics that align with the strengths of B+ trees in OLAP scenarios

- Range Queries: Frequent use of range-based filtering on attributes like dates and discounts.
- Joins: Efficient handling of join operations based on indexed keys.
- Grouping and Aggregation: Effective support for grouping and aggregation operations due to the sorted nature of the index.
- High Selectivity: Queries often focus on specific subsets of data, which B+ trees can efficiently locate and retrieve.

4.1.1 Proof of Successful build

As you can see below is a screenshot of the successful build log using MongoDB's scone build command, we were able to build the MongoDB database from binary without any error.

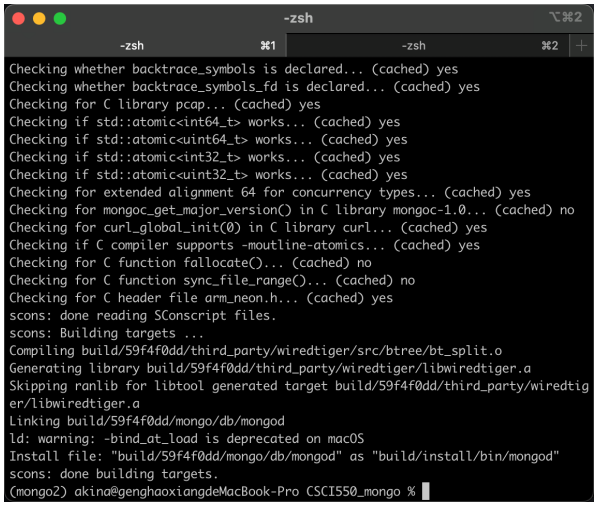

Figure 1: Successful Build Log

4.2 Proof of Correctness

To ensure the correctness of our implementation, we ran the same query in both the original MongoDB and our modified build. For both versions of MongoDB, we used Python PyMongo to connect with the database and run our queries. By juxtaposing the query results in Figures 2 and 3, we can see that we have obtained the same results from both the original and our modified version of the database, proving that our modified version of MongoDB is indeed correct and outputs consistent results.

Figure 2: Query 14 results from the Original MongoDB build

Figure 3: Query 14 results from B+ Tree (our) MongoDB build

4.3 Evaluation

What you see here below are the results from a 10MB dataset generated, Figure 4 below shows the time differences, from this we can see that there is a drop in the running time across all four queries. Figure 5 shows the percentage improvements of these queries, ranging from a 2.3% improvement for Q12 to a massive 25% improvement for q6. for an average improvement of 12.6% for the 10MB dataset.

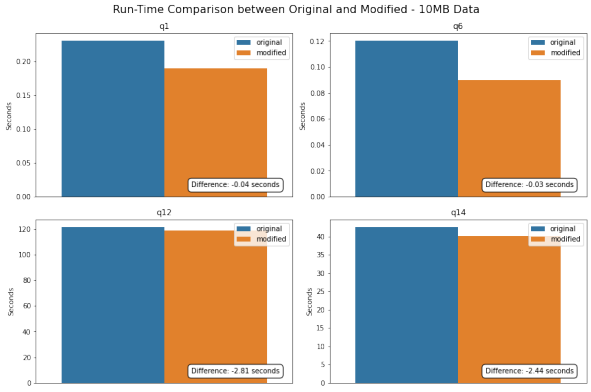

Figure 4: Time Improvements - 10MB

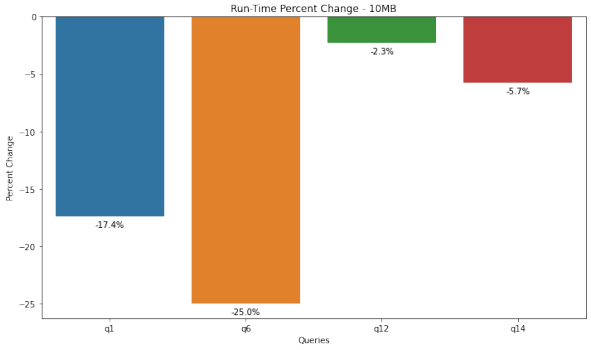

Figure 5: Percent Improvement - 10MB

We also wanted to test our B+ Tree implementation under more stress, so another larger 50MB dataset was used. Figure 6 shows the running time results, we can see that once again there is a drop in the running time across all four queries. Figure 7 shows the percentage differences ranging from a 0.1% reduction for Q14 to again a huge 16.6% reduction for q6, for an average improvement of 6.25% for the 50MB dataset.

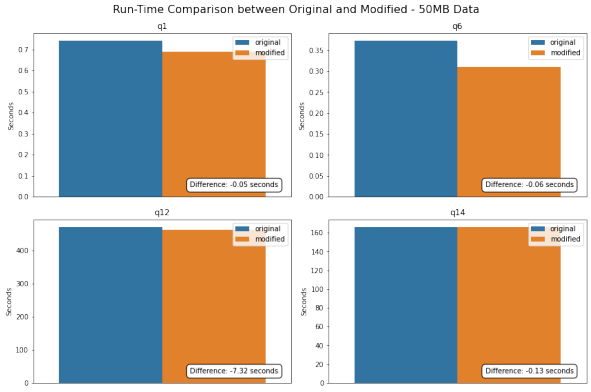

Figure 6: Time Improvements - 50MB

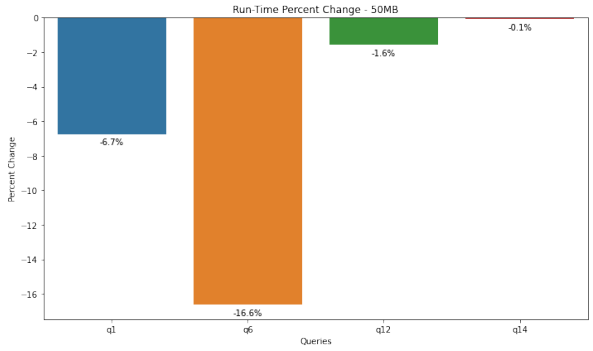

Figure 7: Percent Improvement - 50MB

4.4 Result Analysis

From the above evaluation results, we were curious about why the improvement of adding linked lists in leaf nodes was not that significant at times. To find the answer, we printed out the structure of our B+ tree during our experiment with Figure 8 showing the B+ Tree structure corresponding to the 10MB dataset. The 50MB dataset was identical in terms of the number of levels but just had far more leaf nodes. We found that in both 10MB and 50MB, the B+ Tree only had 2 levels, so when using the original B-tree structure to locate the next page, it would only need to execute one ascend and one descend operation of the cursor, which makes sense because it won't introduce much overhead compared to our new implementation of directly using the next pointer to get the next page.

```
Page Address: 0x6000030f0bd0, Type: Internal (Row-Store), Level: 0
  Child Page Addresses:
    0x6000030fcbd0
    0x6000030fca80
    0x6000030ec690
    0x6000030f0ee0
    0x6000030f0f50
    0x6000030fcaf0
  Page Address: 0x6000030fcbd0, Type: Leaf (Row-Store), Level: 1
  Page Address: 0x6000030fca80, Type: Leaf (Row-Store), Level: 1
  Page Address: 0x6000030ec690, Type: Leaf (Row-Store), Level: 1
  Page Address: 0x6000030f0ee0, Type: Leaf (Row-Store), Level: 1
  Page Address: 0x6000030f0f50, Type: Leaf (Row-Store), Level: 1
  Page Address: 0x6000030fcaf0, Type: Leaf (Row-Store), Level: 1
```

Figure 8: B+ Tree Node Structure of 10MB dataset

In theory, the B+ tree should outperform the B-tree in range scan because there are pointers directly between leaf nodes for faster traversal. However, it might not be suitable for all cases, especially in MongoDB, this is because WiredTiger maintains a shallow-depth B-tree which is pretty unique, which kind of explains why MongoDB sticks with the original default B-Tree for indexing.

5 Conclusions

This project explored the potential performance improvements achievable by integrating a full B+ tree indexing structure within MongoDB. Our investigation revealed that while MongoDB's existing B-tree indexing system provides a solid foundation, it can be further optimized for specific workloads, particularly those involving range scans and sequential access patterns common in OLAP scenarios.

At the same time, although the implementation of a B+ tree demonstrably enhanced performance for the selected TPC-H queries, it does not bring about significant improvement as we previously thought. The reason behind this is that WiredTiger maintains a shallow-depth B-Tree structure, which results in a limited number of ascending and descending operations in search operations. We assume this might be part of the reason that WiredTiger does not extend their B-Tree to B+ Tree.

5.1 Potential Future Goals

The results presented in this report suggest that further investigation and potential integration of B+ tree indexing within MongoDB is warranted, particularly for applications heavily reliant on OLAP workloads. Future research could explore:

- Dynamic Indexing Selection: Developing algorithms to intelligently choose between B-tree and B+ tree indexing based on query patterns and workload characteristics.
- Hybrid Indexing Structures: Exploring hybrid approaches that combine the strengths of both B-trees and B+ trees to address diverse query requirements.
- Performance Evaluation on Real-World Datasets: Expanding testing to encompass real-world datasets from various domains to validate the generalizability of the observed performance improvements.

By pursuing these avenues, we can contribute to optimizing MongoDB's performance and solidifying its position as a leading choice for diverse data management needs.