Database Internal Project

Comparison of Different Secondary Indexing Techniques

Professor

N.L Sarda

Team

Animesh Baranawal-130050013
Rawal Khirodkar-130050014
Ashish Anand-130050035

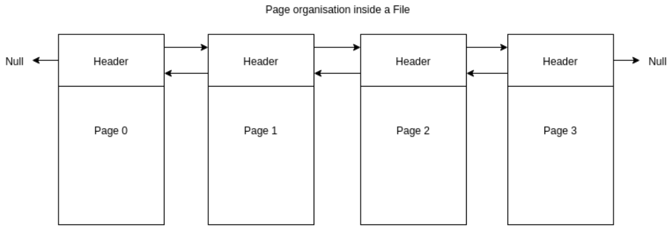
November 5, 2015

Introduction

- In this project we are comparing three different Secondary Indexing Techniques namely:
  - Secondary-key Order Indexing
  - B+ Trees with bucket for pointers corresponding to each Search Key
  - Indexing in B+ tree where modified search key is concatenation with Record Pointer or Serial Number to make it unique.
- Performance Analysis based on execution time of standard operations across the above three mentioned techniques has been done.

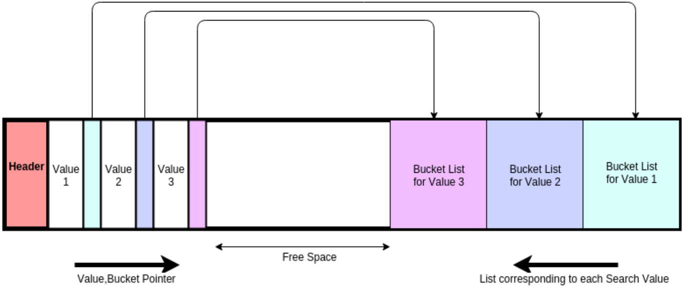
Secondary-Key Order Indexing

- **Basic Data Structure:**
  - Indexing is stored in a file (accessed by filename.index), lets call this as "Index File".
  - Each Index File consists of Pages (accessed by pagenumber). The pages are organised as a doubly linked list inside a file. Assume we have 4 pages inside the file:-



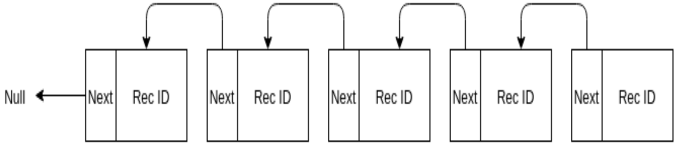
Secondary-Key Order Indexing

- **Basic Data Structure:**
  - A page has a sequential organisation of (Secondary Key, Bucket Pointer) to the left. The bucket pointer points to a bucket contains all the records corresponding to the same secondary key.



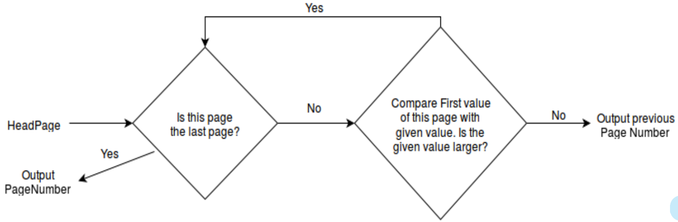
Secondary-Key Order Indexing

- **Basic Data Structure:**
  - The following figure shows the internal structure of a Bucket List. Assume we have 5 records corresponding to same secondary value.



Secondary-Key Order Indexing

- **Search Operation:**
  - Search operation can be broken into two parts :
    - Find the page number where the value can be present
    - Find the index pointer in the page where it can be present
  - Page Number of the page is found by sequentially going through the pages and comparing the first value with the given value.
  - Index Pointer inside the page is found by performing a binary search on all the values inside the page



Secondary-Key Order Indexing

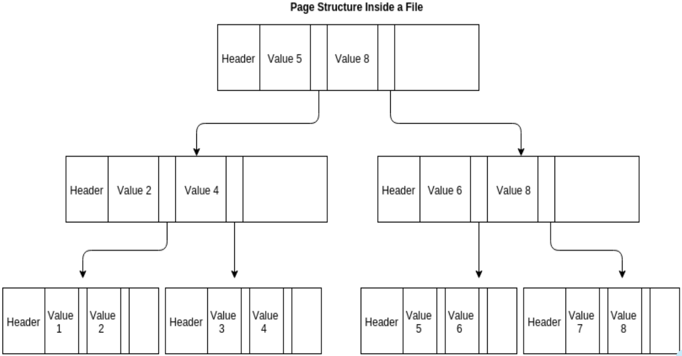
- **Insert Operation:**
  - To insert value, the correct place for insertion is found by searching for the index pointer using the search algorithm.
  - It is then checked if the given page has enough space to insert one value.
    - If enough space is present, the element is inserted by shifting all the values one place to the right and then allocating the space to the given value.
    - If enough space is not present, the page is split into two pages with each pages having half of the values and the insert procedure is started all over again.

Secondary-Key Order Indexing

- **Delete Operation:**
  - The index pointer of the value to be deleted is found out using the search algorithm.
  - The record ID is then found out by traversing over the bucket list of the value.
    - If the record ID is not found, error is thrown.
    - If the record ID is found, bucket list is modified appropriately.
    - It may happen that the bucket list of value becomes empty. In such case, the value is deleted by shifting all the further values by one position.
  - At every deletion step, we check if the page becomes empty.
    - If page becomes empty, the nextpage and previousPage pointers are modified appropriately to maintain the page list structure.
    - Otherwise, nothing is done.

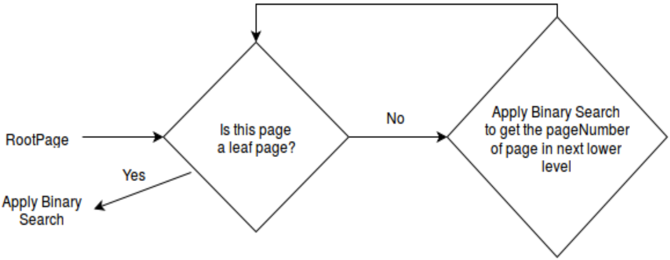
B+ Tree Secondary Indexing

- **Basic Data Structure:**
  - Page organization inside a file is different from Secondary-Key Order Indexing, rest all of the structure of data is same.



B+ Tree Secondary Indexing

- **Search Operation:**
  - If the page is a leaf node, binary search is applied on all values inside the page to give the index pointer.
  - If the page is an internal node, binary search is applied on all values inside the page to give the pagenumber of the page in lower level.



B+ Tree Secondary Indexing

- **Insert Operation:**
  - The index pointer is found out using the search algorithm of the B+ tree.
  - It is then checked if the given leaf page has enough space to accommodate the given value.
    - If space is present, the value is inserted by shifting values inside the leaf page.
    - If space is not present, the leaf page is split into two leaf pages each having half the number of values.
    - The given value is then inserted into one of the two leaf pages based on the index pointer.
    - Appropriate values are forwarded to the parent nodes and the parent nodes are split if they become full.

B+ Tree Secondary Indexing

- **Delete Operation:**
  - The index pointer of the value to be deleted is found out using the search algorithm of the B+ tree.
  - The record ID is then found out by traversing over the bucket list of the value.
    - If the record ID is not found, error is thrown.
    - If the record ID is found, bucket list is modified appropriately.
    - It may happen that the bucket list of value becomes empty. In such case, the value is deleted by shifting all the further values by one position.

Secondary Key + Record ID indexing

- **Basic Idea :**
  - In case of secondary keys, there may be multiple record IDs for each key.
  - Each (secondary Key, recordID) pair is converted to a unique (value,recordID) pair by concatenating the secondary Key with the record ID.
- **Basic Data Structure:** The data structures for both sequential indexing and B+ Tree remain the same. The bucket lists are now of size 1.

Secondary Key + Record ID indexing

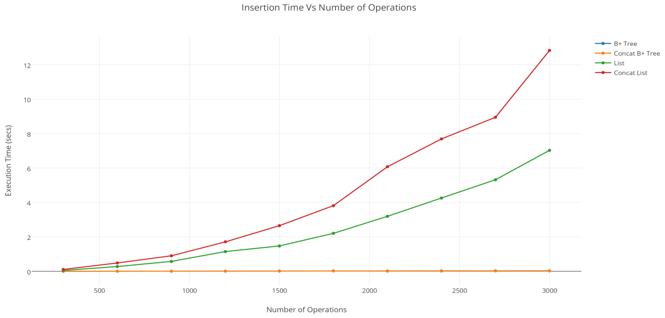
- **Search Operation:** Same algorithm implemented as before for B+ tree and sequential indexing.
- **Insert Operation:** Same algorithm implemented as before for B+ tree and sequential indexing.
- **Delete Operation:** Same algorithm implemented as before for B+ tree and sequential indexing.

Performance Analysis

- Performance Metric used for analysis is "Execution Time" taken for queries.
- The queries consists of standard (insert,delete,search) operations done on four indexing techniques namely
  - B+ Tree
  - B+ Tree with Concatenation of Record ID
  - List form of Secondary Indexing
  - List form with Concatenation of Record ID
- We further classify Search queries as
  - Less than a particular Search Key (lt)
  - Less than equal to a particular Search Key (leq)
  - Greater than a particular Search Key (gt)
  - Greater than equal to a particular Search Key (geq)
  - Equal to a particular Search Key (eq)

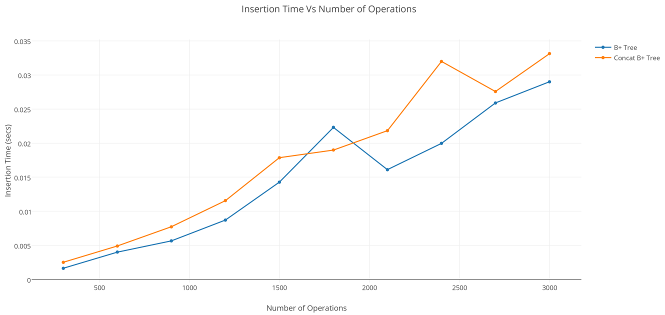
Performance Analysis

- **Insert Operation:**
  - Increase in time with more keys to insert



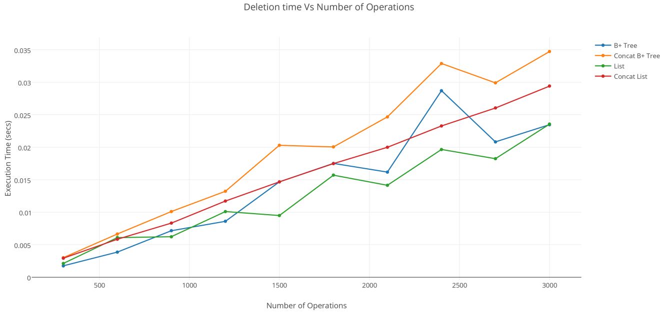
Performance Analysis

- **Insert Operation (Tree Comparison):**
  - more time for insertion in BTree having concatenated indexes



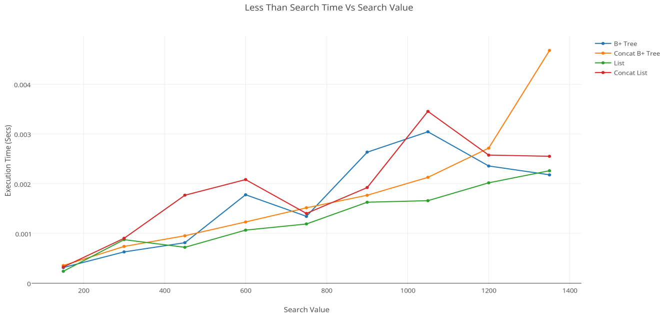
Performance Analysis

- **Delete Operation:**
  - more time for deletion in index structures with concatenated values



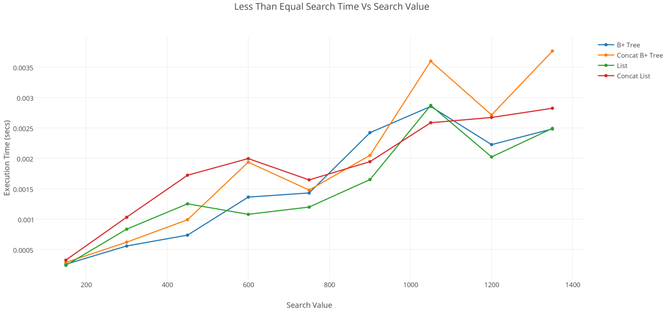
Performance Analysis

- **Search Operation (lt):**
  - More time required for larger values since more records fetched



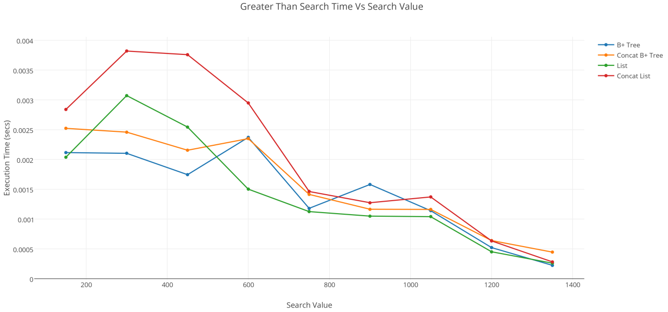
Performance Analysis

- **Search Operation (leq):**
  - More time required for larger values since more records fetched



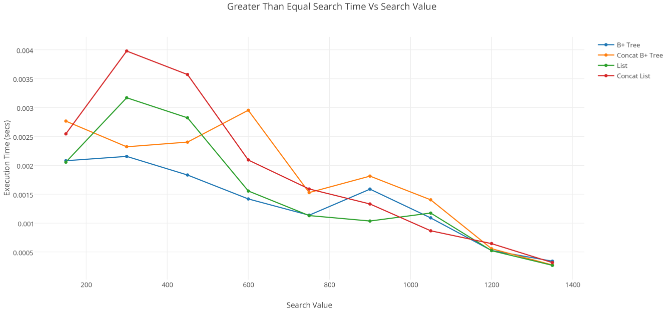
Performance Analysis

- **Search Operation (gt):**
  - Less time required for larger values since less records fetched



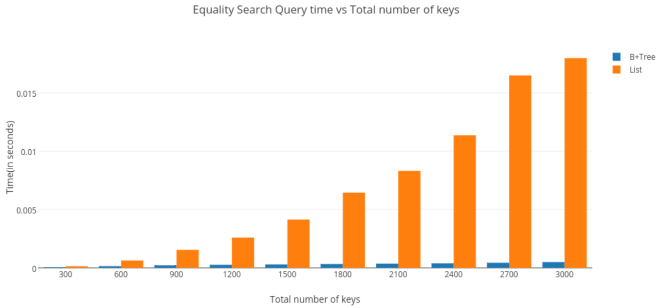
Performance Analysis

- **Search Operation (geq):**
  - Less time required for larger values since less records fetched



Performance Analysis

- **Search Operation (eq):**
  - Search time increases with increase in total number of keys



Conclusion

- List index structure requires more insertions time than B+ tree structure because of the sequential scanning to find correct page number
- Index structures with concatenated search keys require larger insertion time than structure with multiple records due to the extra time required for computing the concatenated values
- As a general trend, more time is required by list structure for range queries than B+ trees

Conclusion

- For range queries, index structures with concatenated search keys require greater time due to extra time required to compute concatenated values.
- For search queries, the trend is very similar to that of insertion because we need to search the index pointer in both the cases. B+ tree search time is much less than List search time.