Designing an Index for ZooDB

Jonas Nick & Bogdan Vancea

June 3, 2014

Contents

1 Introduction

ZooDB is an open source object database written in Java. It is based on the JDO standard, which is a specification of Java object persistence. At this point it lacks some advanced features like concurrency and schema evolution but it has been applied successfully in some university projects. Its selling point is speed on the pole position benchmark.

1.1 Database Index

The following definition captures what we want to understand as a database index. Here is an example where a user of the database creates an index. He has a class Person and wants to retrieve instances of the class rapidly by name. This is called an Attribute Index, which maps values (here names) to Object-IDs ZooDB uses Indexes on several other places, it has for example an ObjectID

index to map OIDs to positions on the disk and an Extension Index to map a Diskposition to the following Diskposition if an Object spans several disk pages.

1.2 B+-Tree

What can be the underlying data structure of an Index? We said that the Index should allow for ordered iteration which makes an ordered data structure such as a search tree ideal. Further, the index is going to be stored in a file on the hard drive, which is why indexes are implemented as B+-Trees This is an example of a B+-Tree, which has an inner node that contains keys and pointer to children and leaf nodes that contain keys and values, which are depicted here as dots. The most important property of a B+-Tree is that a node fills exactly one disk page, which is why the leafs in the example have space here for other keys. This means that a node has a maximum number of entries and in order to use as few disk pages as possible the nodes should be as full as possible, meaning that there is a minimum number of entries that are allowed in a node. This is normally a half of the maximum number of keys.

Here is an example for inserting the entry with key 8 and some value. We see that 8 is smaller than 13, so we have to insert the entry into this leaf. This node would not fit on a disk page anymore so we have to split it in half. And since there are now more children of this parent node, there has to be another key in this node too. So 5 is the key that is copied to the upper level. Now the parent page overflows so the node is split in half and one key goes to the upper level. Leading to this tree, which is balanced and the requirements of minimum and maximum entries are still fulfilled (except in the root).

As we have seen re-balancing on insert takes place by using splits. When deleting an entry and the node becomes underfull, the node is merged with a sibling node. If a merge is not possible, some entries are redistributed from a sibling to the underflowing node. All operations have logarithmic time complexity.

2 Goals & Challenges

We had several goals in this project. ZooDB already had an Index, our reimplementation should be significantly faster. The index implementation has a substantial impact on the performance of the whole database There should be two types of indices, one key unique and one key-value unique. The difference is that in a key unique index the first entry will be overwritten, in a key-value unique index, both entries can exist. The index should support range query iterators Also, the index should support caching using a buffer manager. Basically the buffer manager is responsible for fetching pages from disk and provide them to the B+-tree. And the Index should use prefix-sharing, a technique I am going to explain next.

2.1 Prefix sharing

Imagine this box represents a node which has some keys. The bit-prefix of all keys is the same, so we can store only the deviations from the prefix. You can easily see that we have to store less bits now, meaning that we can pack more keys on a page. Prefix sharing introduces a lot of changes to the

logic of the BTree. With prefix sharing every node can have a variable number of entries, depending on the actual keys. Previously, the maximum number of entries was the same for all nodes. This means that prefix sharing also affects all re-balancing operations.

2.2 Challenges

The runtime of the index is dominated by disk access, therefore having fewer pages meaning fewer nodes is preferred. And since modified nodes have to written to disk, you want to rarely modify nodes. New features are costly, prefix-sharing encoding takes time and caching adds another layer of indirection. Textbook algorithms need to be adapted because they are not optimized for our practical scenario in various ways (sibling pointer, different order). and they do not cover prefix sharing nor dealing with duplicates. The operations for key-value unique trees are in many steps different. And we dealt with low level implementations: replacing multiplications/divisions with bit-shifts and reducing polymorphism

3 Implementation

3.1 Buffer Manager

Explain how all accesses to nodes are done through the BufferManager.

3.2 Class Diagram

For the new implementation of the index we have focused on achieving a balance between good design and code that offers good performance. Figure 1 show the class diagram for ZooDB.

The **BTree** class implements all the logic for the B+ tree operations, including the re-balancing operations. The operations implemented operate on instances of the **BTreeNode** class. The **BTreeNode** abstract class contains the logic for a B+ tree node. To reduce the usage of polymorphism, we have opted not to implement different classes for leaf, inner or root nodes. The **BTreeNode** class has additional boolean fields, *isLeaf* and *isRoot* that are used to describe the position of a node within the tree. This choice reduces the size of the inheritance hierarchy, however in the cases where leaf nodes have to be treated differently than inner nodes we have added additional if statements to separate the behavior.

The **BTreeNode** class holds no information about the child nodes of the node. However, this class exposes a number of abstract child access methods, like *getChild()* or *getChildren()*. These methods are implemented by the **PagedBTreeNode** class. Additionally, the **BTreeNode** class does not hold any information to differentiate between nodes belonging to unique and non-unique trees. All of the methods of for the **BTree** and **BTreeNode** classes which need to have different behaviour for unique and non-unique trees are abstract methods and will be overridden by the **UniquePagedBTreeNode** and **NonUniquePagedBTreeNode** classes.

The **PagedBTreeNode** holds the logic responsible for loading nodes from storage. class extends the **BTreeNode** and contains a reference to the BufferManager associated with the tree. Each instance of **PagedBTreeNode** contains a *pageId*, which is given assigned by the **BufferManager**

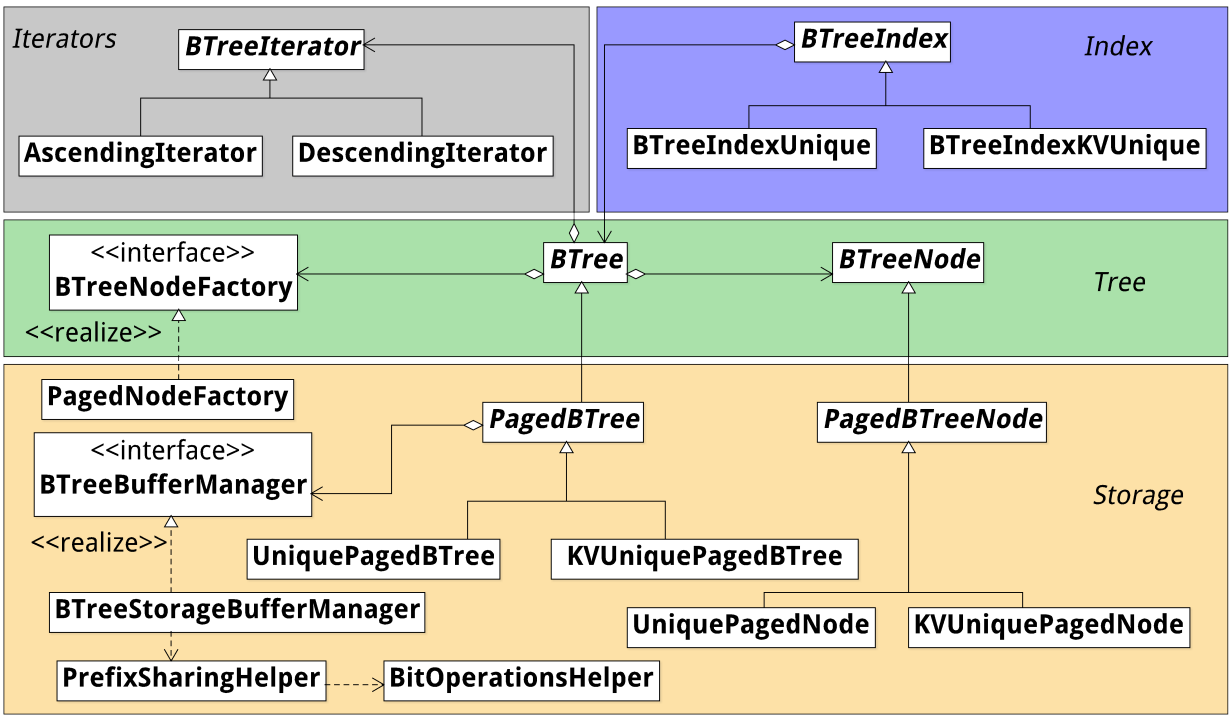

Figure 1: The class diagram for the new index implementation

upon storage. The **PagedBTree** node contains information about the children of a node. We have opted to store the an array consisting of the page ids of the child nodes, instead of an array of hard references to the PagedBTreeNodes. Instead we store an array of weak references to the **PagedBTreeNode** objects. When a new node is loaded from disk, the child array is populated with the page ids of its child nodes while the array of weak references to the children is initialized to an array of null references. The first access to a child node will retrieve the **PagedBTreeNode** instance from the buffer manager and populate the weak references corresponding to its page id. Subsequent accesses to the child will use the weak reference, provided the child objects is not reclaimed by the garbage collector. If any of the children nodes are reclaimed by the garbage collector, they will be retrieved from storage on the next access and the weak reference will be repopulated.

The **UniquePagedBTreeNode** and **NonUniquePagedBTreeNode** classes extend the **PagedBTreeNode** class and contain data and behaviour specific to nodes belonging to unique and non-unique trees. For example, in case the non-unique nodes, inner nodes cotains the both a key array and a value array. This leads to different behaviour for unique and non-unique nodes during operations like peforming binary searches, shifting node entries to the left or right or copying node entries from one node to the other. This different behaviour is implemented in the aforementioned classes: **UniquePagedBTreeNode** and **NonUniquePagedBTreeNode**.

The tree iterators are implemented by the **AscendingBTreeLeafEntryIterator** and **DescendingBTreeLeafEntryIterator** classes. The shared behaviour for these classes is implemented in the abstract class **BTreeLeafEntryIterator**. Because we have opted not to store information about the parent nodes, neither in the form of page ids or references, the iteration process requires the use of

a stack to store the current position in the tree. The iterators do not support **Copy on Write** and are invalidated when the B+ tree is modified.

The indexes are implemented in the **UniqueBTreeIndex** and **NonUniqueBTreeIndex** classes. As the index interface offers both methods to modify the B+ tree and to create iterators, we have opted to separate the tree logic in **BTree** and the iterator in **BTreeLeafEntryIterator**. This means that index objects delete all B+ tree operations to a reference of a **BTree** and create instances of **BTreeLeafEntryIterator** as needed.

The prefix sharing logic has been implemented in the **PrefixSharingHelper** class. Operations like computing the prefix, encoding/decoding the key array using prefix sharing and the re-balancing computations are available as static methods. We have choses to keep all this logic static to avoid creating extra computational objects that would have to be passed to **BTreeNode** objects upon creation.

3.3 B+ Tree Operations

One important implementation choice we have made is to store the key array un-encoded in memory and to perform the prefix sharing encoding only when nodes are written to disk. While this approach increase the size of the node pages when loaded in memory, the advantage of this choice is that the costly key array encoding is only performed once, even if the node is modified multiple times before it is written to disk.

Another advantage of having the key array unencoded in memory is that the search operations is similar to the search operation run on a traditional B+ tree.

For the insert operation, if the new key has been inserted in the first or the last position of the key array, the prefix for the node has to be recomputed. The overflow is handled as follows:

1. The algorithm first attempts to redistribute some keys from the current node to its left sibling. We do this to delay the moment when a new node is created. The exact redistribution process is described in the next section.

2. If that is not possible, the current node is split into 2 nodes of equal size. One important different appears in how the new tree handles keys moving from one node to the other. In a traditional B+ tree, the max number of keys on a node is fixed, which makes it very to perform operations like splitting a node into 2 nodes of equal sizes. In case of prefix sharing, the first half of the keys could have a very different prefix size compared to the second half, which would lead to having 2 nodes with very different sizes. To avoid this, we perform a binary search to compute the optimal number of keys to move to the new node.

In the case of the delete operation, the removal of an entry from the key array can cause the prefix to change. If the node from which the deletion has been performed is now underfull, a re-balancing procedure begin. The steps for the re-balancing are the following:

1. In the first step, the algorithm first checks if the current node can be merged with **either** its left or its right neighbour.

2. If the merge at the previous step is not possible, the next step is to check wether the keys from the current node can be split between its left sibling **and** its right sibling. While this approach

would involve in marking both the left and righ siblings as dirty, the current node can then be removed.

3. If the the previous operation was not possible, some keys are redistribute from either the left or the right sibling. As a note, the success of the rebalancing operations is not guaranteed and the current node could remain underfull after the re-balance operation. However, we observed that this behaviour does not happen very often and does not have a significant impact on the tree size.

3.4 Prefix Sharing Implementation

When storing a node, the key array of a node is encoded into an array of bytes accoding to the common prefix. The encoding was peformed at the bit-level, meaning that both the binary prefix and the deviations of elements from the prefix are computed as arbitrary number of bits. This means that if the binary prefix has length 9, it can be store in 2 bytes. Moreover, the remaining 7 bits from the second byte can be used to store part of the deviation of the fist array element. Other available alternatives for encoding would have been to compute both the prefix and the deviations as multiples of 8 and ensure that each encoded byte contains only prefix bits or deviation bits. We have chosen the bit-level encoding because it offers the best space reduction.

We have based the prefix sharing algorithms on the following observations about the binary prefix for an array of **sorted** integer elements.

- The binary prefix of the array is equal to the binary prefix shared by the first and last element.
- Inserting a new element in the array in an ordered manner will either decrease the prefix of the whole array or leave it unchanged.
- Removing an element from the array will either increase the prefix of the whole array or leave it unchanged.

The main purpose of the split operation is to split the keys of a node that is overflowing between 2 nodes such that the node sizes are roughly the same. We perform this operation using a binary search to determine the optimal number of keys that have be moved from the current node such that the sizes of the current node and new node are as close as possible. The initial value for the number of keys to move is equal to half of the number of keys stored on the current node. The following logic is executed during each round of the binary search.

1. Compute the binary prefixes and sizes for the nodes that would result if the current number of keys k would be moved to a new node.
2. Compute the different between the node sizes and if the current split is the best encountered so far, store the current number of keys to move and the size difference.
3. If the size of the current node after the simulated key move is smaller than the size of the new node, decrease the number of keys to move. Otherwise, increase the number of keys to move.

In the end, the split algorithm returns the number of keys to move corresponding to the mimimum different in size between the current and new node.

A special case of the split operation is when the current node overflow and we would like to check if the key array can be equally split between the left and right siblings. In that case, the binary search is modified to search for number of keys to move to the right node, the remaining keys being moved to the left node. An additional optimization that we have added in that case is to stop at the first split that results in both left and right nodes having a valid size.

The goal of the redistribute operation is to move a number of keys from a node that is not underfull to an underfull node. It is desirable that at the end the split operation, neither of the nodes be underfull. If that is not possible, the redistribution fails and no keys are moved. The number of keys to move to the underfull node is computed in a similar manner to the split operation, using a binary search. Therefore, at each step, the prefixes and the sizes of the node that would result after the operation are computed. The binary search stops at the first split that results into two nodes of valid sizes.

The goal of the merge operation is reduce the number of nodes in the tree by merging an underfull node into either its left or right neighbours. In the case of the prefix sharing, the check that determines if two nodes can be merged is non-trivial, as it involves computing the prefix and the size of the node that would result after the merge. If the new node does not fit into a page file, the merge operation will not be performed.

3.5 Support for different sizes

We have support for this (ex: PagedPosIndex) , maybe we can provide some additional details.