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

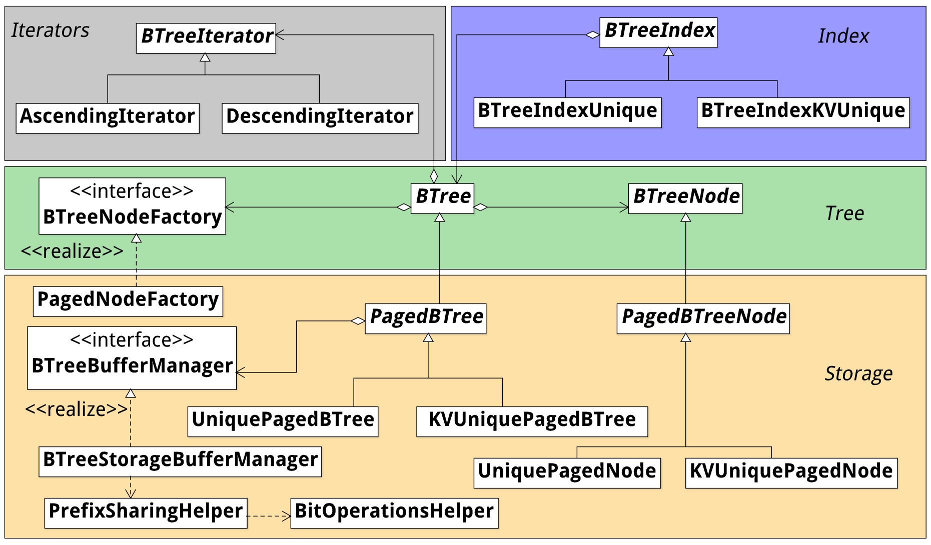
Imagine this box represents a node which has some keys. The bit-prefix of all keys is the same, so we can store only the deviations from the prefix. You can easily see that we have to store less bits now, meaning that we can pack more keys on a page. Prefix sharing introduces a lot of changes to the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



3.2 B+ Tree Operations