Designing an Index for ZooDB

Jonas Nick & Bogdan Vancea

June 6, 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. Up to this point it lacks some advanced features like concurrency and schema evolution but it has already been applied successfully in several university projects. The most important selling point of ZooDB is its speed. It is up to four times faster than competitor db4o on the PolePosition benchmark[1].

1.1 Database Index

A database *index* is a data structure for rapidly retrieving and saving key-value entries. It allows ordered iteration and can be stored in a hard drive file.

The most natural choice for the underlying datastructure of our index are *B+ trees*. It consists of inner nodes which store keys and pointer to children nodes and leaf nodes consisting of keys and values. Each node fills exactly one disk page enabling efficient storage in a file system. Therefore, there is a maximum number of entries that a node can hold – this number is often called *order* of the tree. On the other hand, there is a minimum number of entries that are allowed in a node in order to keep the tree balanced and allocate as few disk pages as possible. This lower bound is normally half the order.

When inserting or deleting an entry, a node might become *over- or underfull* and the tree has to be recursively rebalanced again. If the node is overfull, the node is split into two nodes and the middle key is put into the parent node. After deleting an entry and the node becoming 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. It is important to note that insert, delete and rebalancing operations have logarithmic time complexity.

In ZooDB a user can create an index on an attribute of the object he wants to store to permit fast point and range queries for that attribute. Additionally, ZooDB uses indexes internally at several other places. It uses for example an ObjectID index to map OIDs to positions on the disk and an Extension Index to map a disk position to the following disk position if an object spans several disk pages.

2 Goals & Challenges

ZooDB already had a working index implementation, the goal was that the reimplementation should be significantly faster. Since indexes have multiple roles in the database, the speed of the implementation has a substantial impact on the performance of the whole database. The runtime of the index is largely influenced by disk access, therefore the implementation should minimize the number of nodes. Also, nodes should be rarely modified, because in that case they have to be written to disk.

Additionally, the index should support caching via a *buffer manager*. Basically, the buffer manager is responsible for fetching pages from disk and providing them to the tree. This allows not loading

---

[1]http://polepos.org/

the whole tree into the memory, but restricting the cache to a fixed size and load and unload pages as needed.

There should be two types of indexes, one is called *key unique* and the other *key-value unique* (or non-unique). Consider for example inserting first the key-value entry $(1, 1)$ and subsequently inserting $(1, 2)$. The difference is that in a key unique index the first entry will be overwritten, but in a key-value unique index both entries remain stored.

Unfortunately, the new features are costly: prefix-sharing encoding takes some additional time and caching adds another layer of indirection. In order to get the best performance out of the code, we dealt with low level implementation optimizations like replacing multiplications and divisions with bit-shifts and reducing polymorphism.

Textbooks do a good job of explaining the big picture of the B+ tree operations but miss some implementation details. For example, in their description leafs are linked through sibling pointers. Since ZooDB is a copy-on-write database, sibling pointers are not possible. Pages are seldomly stored on the same place twice, so if you store leaf l_1, you have to update the sibling pointer of l_2, store l_2, update the sibling pointer of l_1, store l_1 and so on.
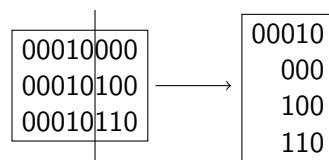
2.1 Prefix sharing

Figure 1: The concept behind prefix sharing

From the start on, the most important idea to speed up the reimplementation was to make use of *prefix sharing*. Imagine the left box in figure 1 represents a node which has some keys shown as bit strings. All keys in the node have a prefix in common, so the same information is contained if we store the prefix and the additional bits (box on the right). It is easy to see that less bits have to be stored now and therefore more keys can be packed on a page. However, prefix sharing entails a lot of changes to the operations of the B+ tree. Because the prefix, and specifically its length, depends on the actual keys in a node, the maximum number of entries is different for every node. This means that prefix sharing also affects all rebalancing operations.

3 Implementation

3.1 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 2 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
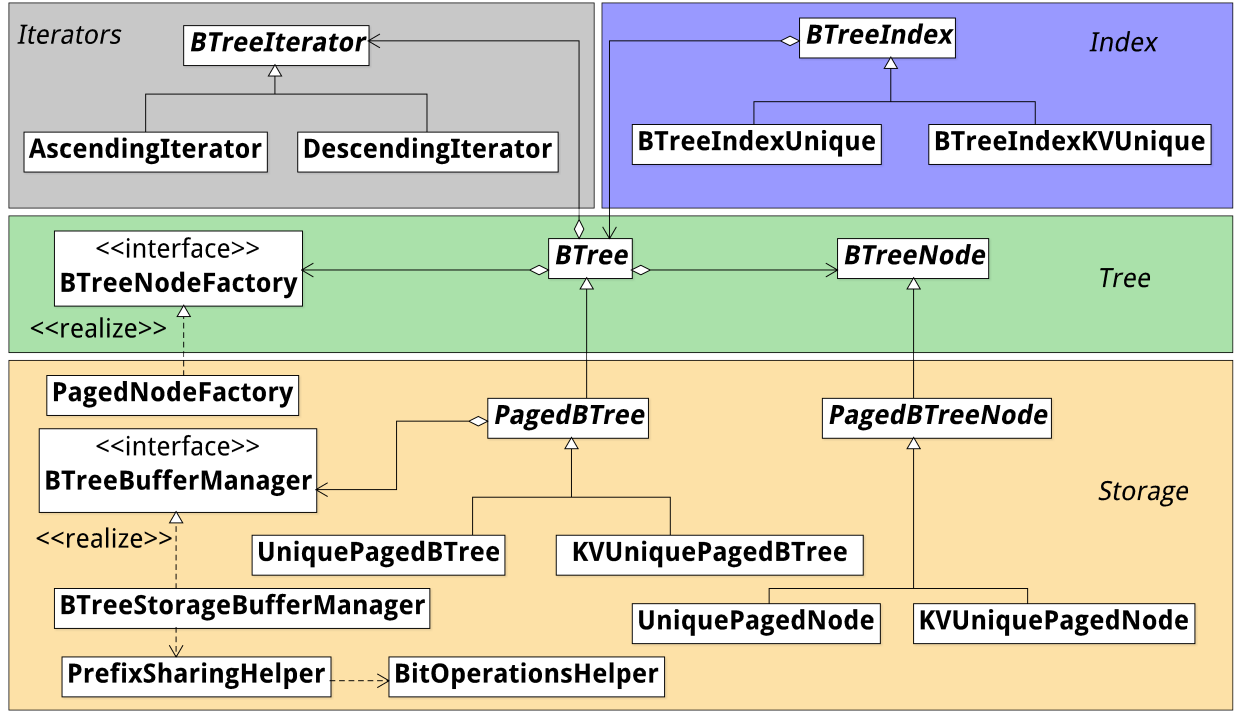
Figure 2: The class diagram for the new index implementation

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** 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

4

from the **BufferManager** 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 **PagedB-TreeNode** 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 **Descending-BTreeLeafEntryIterator** 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.

3.2 Buffer Manager

The buffer manager is responsible for reading and writing **PagedBTreeNodes** to the **StorageChannel**.

The buffer manager instance is created in the constructor of a **BTreeIndex**. Its only direct function for the index is to ensure that all its content is written to disk when `write` is called on the index.

The buffer manager has two hash maps that map page ids to **PagedBTreeNode**s, one is for storing dirty (`dirtyBuffer`), the other for clean nodes (`cleanBuffer`). Using two hash maps allows to enforce a cache size on the `cleanBuffer` because its entries can be safely cleared.

The constructor of **PagedBTreeNode**s takes a **BufferManager** as argument and calls `save` on the buffer manager. This advises the buffer manager to store the node in the `dirtyBuffer` with a negative id to indicate that this node has not been written yet. If the index calls `write` on the buffer manager, it traverses through the tree, writes all dirty nodes to storage and marks them as clean. During `write`, the buffer manager allocates a page, encodes a nodes data to a writable format and applies prefix sharing.

If a node's status changes from dirty to clean or vice versa, the node calls `updatePageStatus` on the buffer manager, which moves the node from the `dirtyBuffer` to the `cleanBuffer` or the other way round.

A **PagedBTreeNode** stores its children as an array of page ids, so the node needs the buffer manager to resolve page ids to actual **PagedBTreeNode** instances by calling the buffer manager's `read`. Then the buffer manager tries to look up the page id in one of its memory buffers and if the page id is not in one of the memory buffers, the storage channel is queried.

Whenever a **PagedBTreeNode** is removed from the tree, it calls `close` on itself, which in turn calls `remove` on the buffer manager. This instructs the buffer manager to delete the node from its memory buffers and to free the corresponding page.

3.3   B+ Tree Operations

One important implementation choice we have made is to store the key array unencoded 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 whether 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 right siblings as dirty, the current node can then be removed.

3. If 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 according to the common prefix. The encoding was performed 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 minimum 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.

4   Results

4.1   Microbenchmarks

These microbenchmarks compare the old index and the reimplementation on artificial data. Each parameter configuration was rerun ten times. The page size was set to 4096 bytes.

random is the worst case scenario show worst case if we are better, best case if we are worse search is searching each element * non-unique tree is 10 times duplication

* best case: "non-unique-increasing-insert"

* search? our index worse for nonUniqueIndex-increasing? certainly worse for unique-random, unique increasing worse as well (WHY?),

* after remove our tree is more balanced (random, unique, because that was before not so well balanced) * time of delete not so good increasing for unique much! faster in general

4.2   StackOverflow Import

We have tested the performance of the new index implementation on a real world workload consisting of importing StackOverflow data from an XML dump. This data consisted of: 1.3 million users, 10.3 million posts, 13 million comments and 25 million votes. The attribute indexes that have been used for these objects were: 3 key unique indexes and 9 key-value unique indexes. The page size used was 4KB and the final database size was approximately 31 GB.

Figure 12 shows the comparison between the number of index pages for different types of database indexes after the import operation was completed. The free space index is not included in the graph as it did not use the prefix sharing implementation. The attribute index was 41.6% smaller, the OID index was 41.5% small and the Position index was 23.1% smaller. The total reduction of index pages was 38.5%.
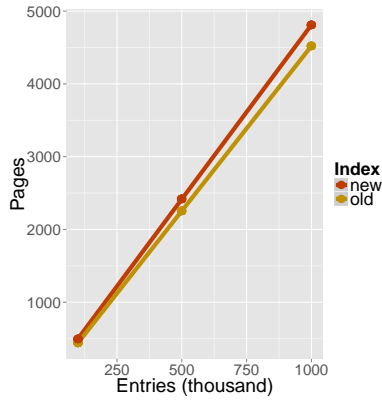
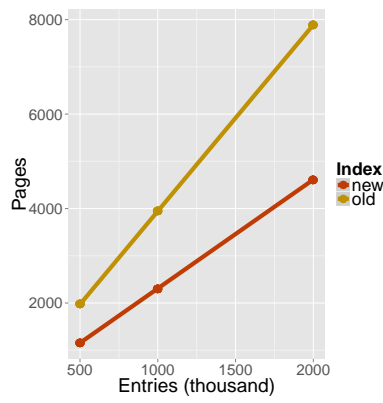Figure 3: Insert random entries in a key unique index.

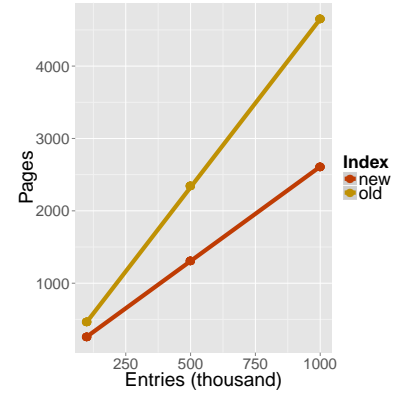Figure 4: Insert increasing entries in a key unique index.

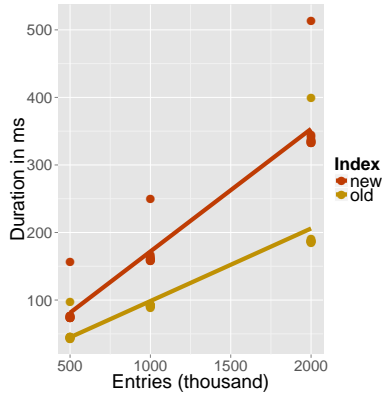Figure 5: Insert random entries in a key-value unique index.

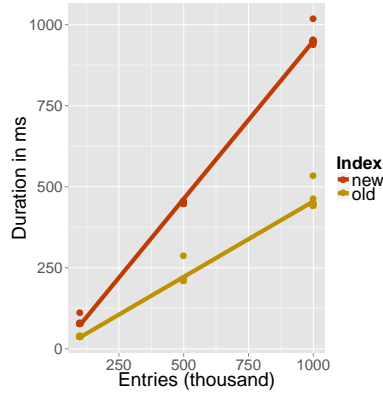Figure 6: Insert increasing entries in a key unique index.

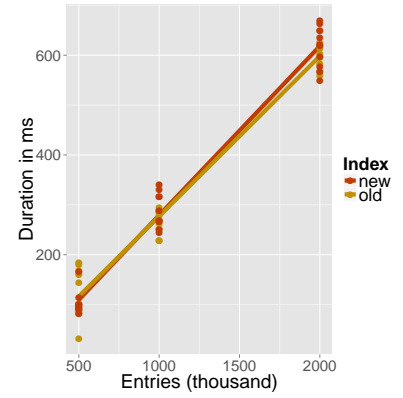Figure 7: Insert random entries in a key-value unique index.

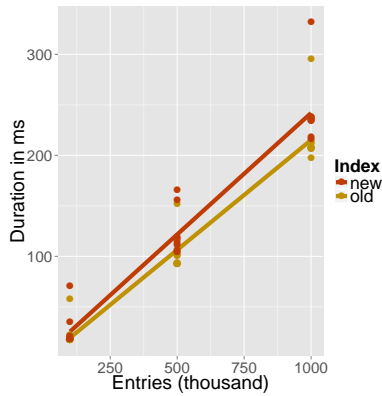Figure 8: Write after inserting increasing entries in a key unique index.

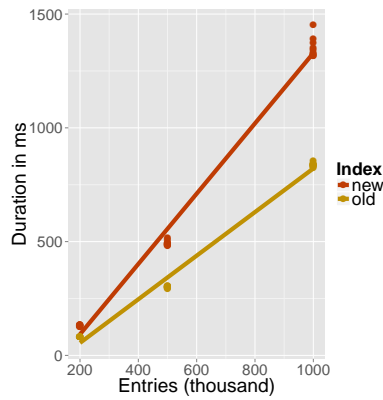Figure 9: Search increasing entries in a key unique index.

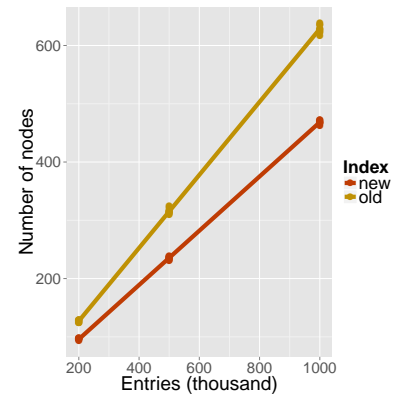Figure 10: Remove increasing entries from a key unique index.

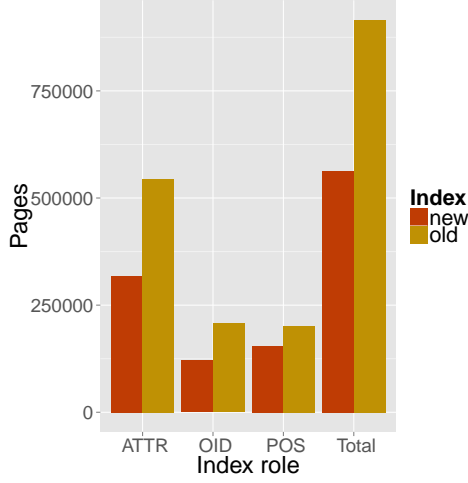Figure 11: Remove random entries from a key unique index.

9

Figure 12: Comparison of number of index pages for the two index implementations
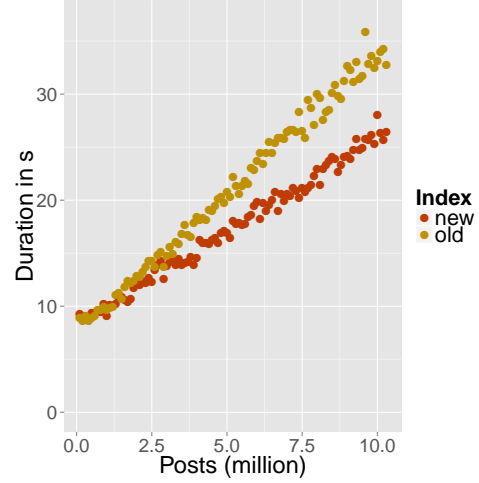
Figure 13: Evolution of commit times for post objects

Figure 13 shows the evolution of the commit durations for post objects. Each point on the graph represents the commit of a batch of post objects. For each point, the distance on the horizontal axis represents the number of post objets already stored in the database, while the distance on the vertical axis shows the commit duration. One can notice that for both indices, the commit duration increase as the number of posts that are already stored increases. However, the prefix sharing index presents a slower increase, as the commit times are generally smaller than the commit times for the old index implementation.

Moreover, the total duration of the import with the new index implementation was 25% smaller than the duration of the import with the old index. The main reason for this difference is that the number of pages writes is smaller for the new index implementation. As the number of entries stored on a page increases, there is a higher probability that all writes in a transaction are restricted to a smaller number of pages, especially since the data accesses in a database respect a certain data locality.