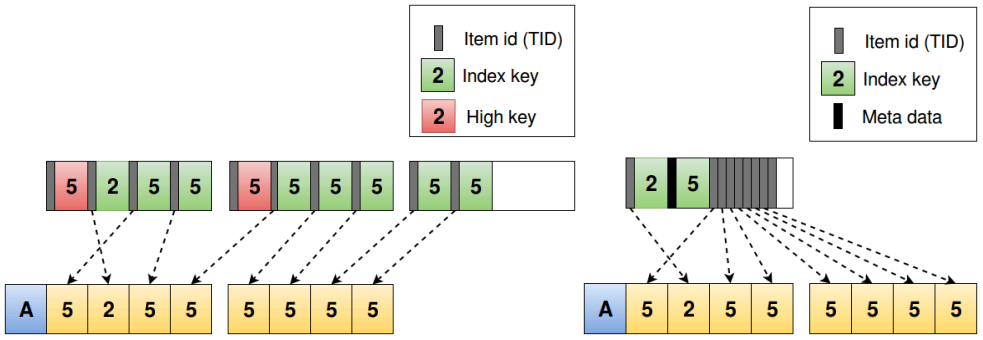
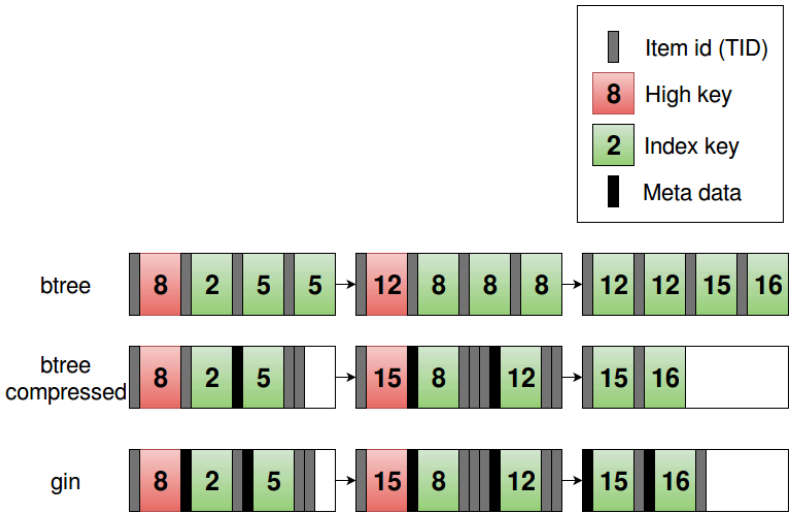
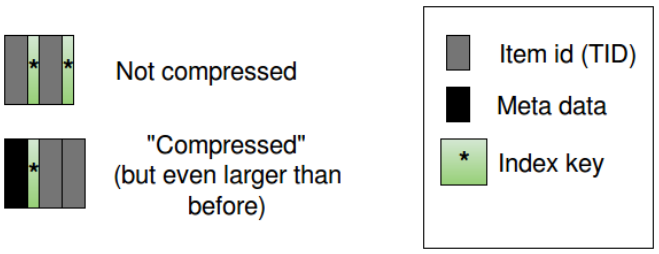
Compression. To be correct, it's not actually compression, but just effective layout of ItemPointers on an index page.
compressed tuple = IndexTuple (with metadata in TID field+ key) + PostingList



1. Gin index fits extremely good for really large sets of repeating keys, but on the other hand, it completely fails to handle unique keys. To btree it is essential to have good performance and concurrency in any corner cases with any number of duplicates. That's why we can't just copy the gin implementation of item pointers compression. The first difference is that btree algorithm performs compression (or, in other words, changes index tuple layout) only if there's more than one tuple with this key. It allows us to avoid the overhead of storing useless metadata for mostly different keys (see picture below). It seems that compression could be useful for unique indexes under heavy write/update load (because of MVCC copies), but I don't sure whether this use-case really exists. Those tuples should be deleted by microvacuum as soon as possible. **Anyway, I think that it's worth to add storage_parameter for btree which enables/disables compression for each particular index. And set compression of unique indexes to off by default. System indexes do not support compression for several reasons. First of all because of WIP state of the patch (debugging system catalog isn't a big pleasure). The next reason is that I know many places in the code where hardcode or some non-obvious syscache routines are used. I do not feel brave enough to change this code. And last but not least, I don't see good reasons to do that.**

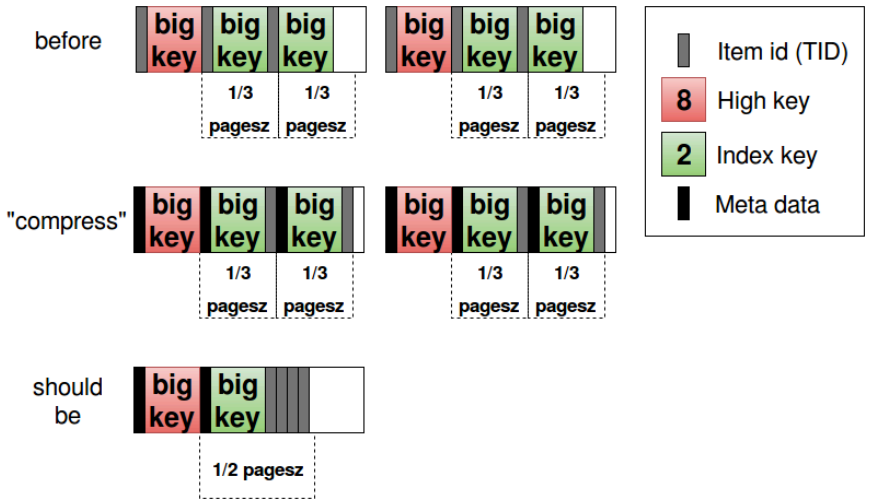


2. If the index key is very small (smaller than metadata) and the number of duplicates is small, compression could lead to index bloat instead of index size decrease (see picture below). **I don't sure whether it's worth to handle this case separately because it's really rare and I consider that it's the DBA's job to disable compression on such indexes. But if you see any clear way to do this, it would be great.**



3. For GIN indexes, if a posting list is too large, a posting tree is created. It proceeded on assumptions that:
    a. Indexed keys are never deleted. It makes all tree algorithms much easier.
    b. There are always many duplicates. Otherwise, gin becomes really inefficient.

    In btree we can't afford these assumptions. So we should handle big posting lists in another way. If there are too many ItemPointers to fit into a single posting list, we will just create another one. The overhead of this approach is that we have to store a duplicate of the key and metadata. It leads to the problem of big keys. If the keysize is close to BTMaxItemSize, compression will give us really small benefit, if any at all (see picture below).



4. The more item pointers fit into the single posting list, the rare we have to split it and repeat the key. Therefore, the bigger BTMaxItemSize is the better. The comment in nbtree.h says: "We actually need to be able to fit three items on every page, so restrict any one item to 1/3 the per-page available space." That is quite right for regular items, but if the index tuple is compressed it already contains more than one item. Taking it into account, we can assert that BTMaxItemSize ~ ⅓ pagesize for regular items, and ~ ½ pagesize for compressed items. **Are there any objections? I wonder if we can increase BTMaxItemSize with some other assumption? The problem I see here is that varlena highkey could be as big as the compressed tuple.**

5. CREATE INDEX. _bt_load. The algorithm of btree build is following: do the heap scan, add tuples into spool, sort the data, insert ordered data from spool into leaf index pages (_bt_load), build inner pages and root. The main changes are applied to _bt_load function. While loading tuples, we do not insert them one by one, but instead, compare each tuple with the previous one, and if they are equal we put them into posting list. If the posting list is large enough to fit into an index tuple (maxposting size id computed as BTMaxItemSize - size of regular index tuple) or if the following tuple is not equal to the previous, we should create packed tuple using BtreeFormPackedTuple on posting list (if any) and insert it into a page. The same we do if there are no more elements in the spool.

6. High key is not a real data, but just an upper bound of the keys that allowed on the page. So there's no need to compress it. While copying a posting tuple into a high key, we should to get rid of posting list. A posting tuple should be truncated to length of a regular tuple, and the metadata in its TID field should be set with appropriate values. It's worth to mention here a very specific point in _bt_buildadd(). If current page is full (there is no room for a new tuple), we copy the last item on the page into the new page, and then rearrange the old page so that the 'last item' becomes its high key rather than a true data item. If the last tuple was compressed, we can truncate it before setting as a high key. But, if it had a big posting list, there will be plenty of free space on the original page. So we must split Posting tuple into 2 pieces. see the picture below and comments in the code. **I'm not sure about correctness of locking here, but I assume that there are no possible concurrent operations while building index. Is it right?**



7. Another difference between gin and btree is that item pointers in gin posting list/tree are always ordered while btree doesn't require this strictly. If there are many duplicates in btree, we don't bother to find the ideal place to keep TIDs ordered. The insertion has a choice whether or not to move right. Currently, we just try to find a page where there is room for the new key. **The next TODO item is to keep item pointers in posting list ordered. The advantage here is that the best compression of posting list could be reached on sorted TIDs. What do you think about it?**

8. Insertion. After we found the sutable place for insertion, check, whether the previous item has the same key. If so, and if there is enough room to add a pointer into the page, we can add it into item. There are two possible cases. If old item is a regular tuple, we should form new compressed tuple. Note, that this case requires to have

enough space for two TIDs (metadata and new TID). Otherwise, we just add the pointer into existing posting list. Then delete old tuple and insert the new one.

9. Search. Fortunately, it's quite easy to change search algorithm. If compressed tuple is found, just go over all TIDs and return them. If an index-only scan is processed, just return the same tuple N times in a row. To avoid storing duplicates in currTuples array, save the key once and then connect it with posting TIDs using tupleOffset. It's clear that if compression is applied, the page could contain more tuples than if it has only uncompressed tuples. That is why MaxPackedIndexTuplesPerPage appears. Array items (which actually has currTuples and tupleOffset) in BTScanPos is preallocated with length = MaxPackedIndexTuplesPerPage, because we must be sure that all items would fit into the array.

10. Split. The only change in this section is posting list truncation before insert the tuple as a high key.

11. Vacuum. Check all TIDs in a posting list. If there are no live items in the compressed tuple, delete the tuple. Otherwise do the following: form new posting tuple, that contains remaining item pointers; delete "old" posting; insert new posting back to the page. One more issue here. If all but one items were deleted from posting list, we should rewrite the item from posting to normal tuple. Microvacuum of compressed tuples is not implemented yet. It's possible to use high bit of offset field of item pointer to flag killed items. But it requires additional performance testing.

12. Locking. Compressed index tuples use the same functions of insertion and deletion as regular index tuples. Most of the operations are performed inside standart functions and don't need any specific locks. Although this issue defenitely requires more properly testing and review. All the operations where posting tuple is updated in place (deleted and then inserted again with new set of item pointers in posting list) are performed with special function _bt_pgupdtup(). As well as operation, where we want to replace one tuple with another one e.g. in btvacuumpage() and _bt_buildadd (see issue related to high key).

13. Xlog. Done. Please, read comments.